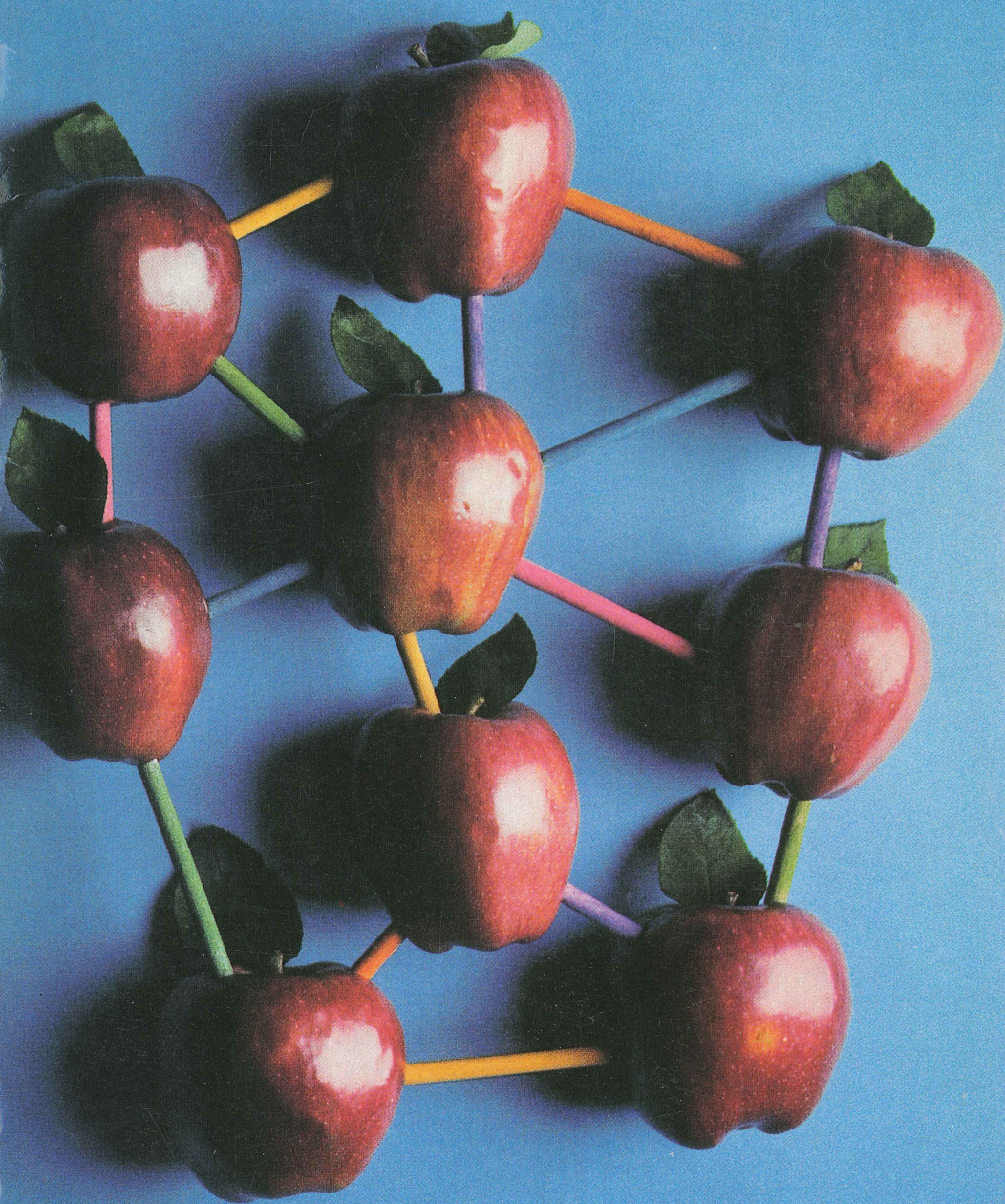


# INTERMEDIATE-LEVEL APPLE II® HANDBOOK

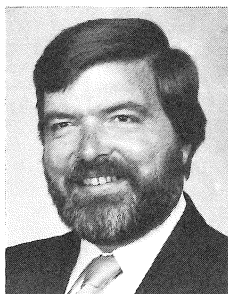
DAVID L. HEISERMAN





# Intermediate-Level Apple II® Handbook

**David L. Heiserman** has been a freelance writer since 1968 and is the author of more than 100 magazine articles and 17 technical and scientific books. He studied applied mathematics at Ohio State University. He is especially interested in the history and philosophy of science.





# Intermediate-Level Apple II® Handbook

by

David L. Heiserman

**Howard W. Sams & Co., Inc.**  
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

Copyright © 1983 by David L. Heiserman  
Indianapolis, IN 46268

FIRST EDITION  
FIRST PRINTING—1983

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-21889-5  
Library of Congress Catalog Card Number: 82-61963

Edited by: *Richard Krajewski*  
Illustrated by: *Kevin Caddell*

*Printed in the United States of America.*

# Preface

The Apple II®\* computer is a marvelous machine. It compares quite favorably with other brands of personal computers on the market today in terms of performance and cost. And it's an easy machine to use. In the jargon of this business, it is "user friendly." However, that notion applies only as long as you stay on the beaten path. Attempt to get away from either your own simple BASIC programs or ready-made programs, and you are bound to run into a bit of difficulty.

The difficulty stems from the flexibility of the Apple. It is this great flexibility that is responsible for the lengthy and sometimes confusing dissertations in the Apple operating manuals.

Most people who are buying and using Apple computers these days are not aware of how to deal with the full potential of such a highly flexible machine. Beginners prefer their instructions to be written in a clear-cut ABC fashion: do this and this, and this is going to happen. That works out quite well as long as the user is doing the simpler, more popular programming operations. But attempt to get away from the simple and popular ideas, and things get a lot trickier.

The Apple is so flexible that an ABC approach teaches the user very little. A wider approach is offered in the technical manuals but it is too general. Indeed, it is suitable only for those users who are both well acquainted with the Apple and well versed in the fundamentals of computer technology. Unfortunately, that doesn't describe most users.

The purpose of this book is to fill in the gap between the ABC approach and technical manual approach. It will lead you very gradually from the usual, popular way of doing things with the Apple into an environment where careful thinking and planning is more important than the mechanics of actually executing a program. It is an environment that promises a great deal of satisfaction, but only at the cost of having to learn new ideas and experiment with them on your own.

If you have already mastered the fundamentals of BASIC program-

---

\*Apple II is a registered trademark of Apple Computer, Inc.

ming and want to do more—a lot more—with your Apple, this is the book for you.

Study the material carefully and run the recommended programs. The programs in the book were prepared for an Apple II with ROM-based Integer BASIC. Equally important, try devising applications of your own as you go along. Above all, learn and have fun doing it.

DAVID L. HEISERMAN



# Contents

## Chapter 1

YOU, YOUR APPLE II, AND THIS BOOK .....	11
Which System Do You Need?—How This Book Is Organized— How To Get the Most From This Book	

## Chapter 2

DISPLAYING TEXT .....	17
The Standard Text Format—Role of the Cursor—Controlling Cursor Position With PRINT Statements—Setting the Cursor Position With TAB Statements—Working With the Cursor- Position Registers—More Cursor-Related Operations	

## Chapter 3

ALTERNATIVE TEXT FORMATS .....	43
Alternative Character Formats—Alternative Print Windows	

## Chapter 4

POKE TO VIDEO MEMORY .....	63
Organization of the Text Memories—Video Character Codes— Getting Some Help From the Monitor—Building and Using Mes- sage Blocks—Working With the Secondary Text Page	

## Chapter 5

THE KEYBOARD ENVIRONMENT .....	85
Supplying Information With INPUT—Controlling Program Flow With INPUT—Strobing the Keyboard With PEEK State- ments—Single-Keystroke Control of a Program—Decoding Single Keystrokes for Control Purposes	

## Chapter 6

THE LOW-RESOLUTION GRAPHICS ENVIRONMENT .....	111
The Elementary Principles—POKEing Colors to the Screen— Alternative Screen Formats	

## Chapter 7

THE HIGH-RESOLUTION GRAPHICS ENVIRONMENT .....	141
Reckoning With LOMEM and HIMEM—Initializing the Hi-Res System—High-Resolution Colors and Screen Format—Doing Some High-Resolution Graphics—Alternative Hi-Res Screen Formats—Hi-Res Shape Tables—Hi-Res Video Addresses	

## Chapter 8

USING SHORT MACHINE-LANGUAGE ROUTINES WITH BASIC .....	195
A Few Useful Machine Instructions—Entering and Running Machine-Language Routines—Calling Some Monitor Rou- tines—Passing Variables to a Machine-Language Routine— Passing Variables From a Machine-Language Routine	

## Chapter 9

THE MEMORY ENVIRONMENT .....	225
Lower RAM Addresses \$0000 Through \$0BFF—Upper RAM Addresses \$0C00 Through \$BFFF—I/O Addresses \$C000 Through \$CFFF—Main ROM Addresses \$D000 Through \$FFFF	

**Chapter 10**

**PROGRAMMING WITH THE MINIASSEMBLER . . . . . 255**  
    A First Look at Some Assembly-Language Programming—Using  
    the Miniassembler—Preparing Assembly-Language Programs—  
    Loading Through the Monitor—Debugging With the BRK In-  
    struction

**Appendix A**

**NUMBER-SYSTEM BASE CONVERSIONS . . . . . 269**

**Appendix B**

**CHARACTER CODES FOR TEXT PRINTING OPERATIONS . . . . . 277**

**Appendix C**

**ORGANIZATION OF THE TEXT/LOW-RESOLUTION GRAPHICS  
VIDEO MEMORY . . . . . 280**

**Appendix D**

**CODES GENERATED BY KEYSTROKES . . . . . 282**

**Appendix E**

**LOW-RESOLUTION GRAPHICS COLORS . . . . . 286**

**Appendix F**

**RANGE OF HIGH-RESOLUTION GRAPHICS VIDEO ADDRESSES . . . . . 301**

**Appendix G**

**6502 INSTRUCTION SET . . . . . 313**

**Index . . . . . 319**





# You, Your Apple II, and This Book

Do you remember the first day you fired up your brand-new Apple Computer? You probably do. For most of us that was an exciting and rewarding experience.

1

One of the nice things about owning your own home computer is that the feeling of having first-time adventures doesn't have to wear off; there is always something new to learn and try. Learning something new, trying it, and making it work can be just as much fun as turning on the computer for the first time.

Certainly there are times when things don't go right and you feel like throwing the whole system across the room. Things go wrong, they don't work out as expected, and the frustration level grows to disheartening proportions. But that happens to everyone who works with computers and computer programming, no matter how much or how little experience they have and no matter how sophisticated or modest the computer system might be.

Home computer programming, however, still retains all the potential for being a continuously rewarding experience. All you have to do is learn what you need to know as you go along, and apply the new found knowledge until it becomes second nature to you. Then you are ready to learn something else. There is really no end to it. And it's great fun.

The key to maintaining an ongoing love affair with your computer is in learning to do new things with it. Doing the same old things in the same old fashion can become boring or tedious, no matter how well they work. But there is excitement in learning.

The primary objective of this book is to help you get more fun out of creating computer programs on your Apple II. The idea is to help you engage in that unending and rewarding adventure called learning.

How does this book help you? Basically, it describes some powerful operating details that are usually mentioned too briefly or overlooked al-

together in the standard user's manuals. These operating details make it possible for you to take advantage of the flexibility of your computer. There are plenty of examples and demonstrations to illustrate the operating details, but you are the one who will have all the fun of putting the details to work in your own programs.

**WHICH SYSTEM DO YOU NEED?** Apple computers are now available in such a wide variety of configurations that it is impractical to attempt writing a book that suits all of them equally well. It is thus necessary to draw some lines, meeting the needs of the largest number of readers and hoping that others will find information that is useful to them and applicable to their system configurations.

The examples and demonstrations in this book have been worked around an Apple II having 48K of RAM- and ROM-based Integer BASIC. However, you do not need a full 48K of RAM to use this book. A 16K system will work quite well except with the material dealing with the secondary page of high-resolution graphics.

The discussions generally apply equally well to cassette- or disk-based systems. Disk operating systems (DOS) can cause some problems at times, especially where DOS boots up in sections of RAM that serve other purposes. DOS users will have to consult their technical manuals to discover ways to resolve any conflicts in RAM organization.

A lineprinter can be helpful in some of the discussions, but it is never a critical requirement.

The discussions of color graphics require a color TV receiver or monitor, but a black-and-white unit will do the job. Incidentally, the color names used throughout the book are the same as those used by the Apple company. Your interpretation of the colors might be slightly different.

**HOW THIS BOOK IS ORGANIZED** When most people are introduced to their first home computer, they get a lot of delight from running BASIC programs listed in the user's manuals and, maybe, running a few "canned" programs—usually game programs.

But we have to be honest here: Running simple BASIC programs and prepared cassette or disk programs can wear thin after a while. Of course you can buy more elaborate and expensive prepared programs, or start entering some programs from published BASIC or machine-language listings; but even if they meet your expectations (and many of them won't), they also become old hat after a while.

One way to overcome this sagging enthusiasm for your computer is to begin writing your own programs. That can be a lot of fun, especially if you

know what you're doing. Learning to use conventional BASIC can keep you going for a long time.

Indeed, writing custom programs in BASIC can serve a lot of personal needs; however, it usually doesn't take long to become dissatisfied with the limitations of BASIC. Experienced Apple BASIC programmers often begin feeling straitjacketed by some of the built-in procedures. It is quite possible to know exactly what you want to do but find that BASIC cannot handle the job as effectively or adequately as you'd like.

Animated graphics, for instance, can fall flat in BASIC because of the long execution times some BASIC statements have. Assigning more than 250 characters to a single string variable also causes problems in BASIC. In such instances, the built-in relationships between BASIC and the hardware system stand in your way.

There are many instances where the structure of the system and the way BASIC works serve as roadblocks to effective programming.

One way to tackle this problem is by digging through the avalanche of books and magazines written for people who want to get around the limitations of their present know-how. If you have tried that route, you have probably been disappointed more than once. It isn't that there is anything necessarily wrong with the available information; it can prove quite valuable in many ways. But most of the literature dealing with Apple tricks and techniques is very specific; they apply only to the particular situation the author is describing. With such literature, you run the risk of missing the principle behind the technique.

More often than not, the real value of a book or article lies in the principles and gems of wisdom that are tucked away in the program listings or accompanying text. Specific solutions for specific problems may or may not be truly helpful, but the methods and ideas behind them can be invaluable.

For example, an article describing how to move a colored spot of light across the screen by depressing a certain key might not seem all that useful or exciting to you; but the technique for sensing key depression or drawing the moving spot of light can be applied in countless ways, once you grasp the main principles behind those actions.

This is a book about main principles. You won't have to dig through the program listings to uncover important ideas; they are clearly spelled out in each case.

Yes, indeed, there are a lot of program listings in this book, but they are intended only to illustrate the workings of the principles at hand. The programs, by themselves, aren't all that useful or exciting. Instead, they are to-the-point illustrations and not highly refined, fully developed programs. They are trimmed to the bare bones so that the point they illustrate will stand out as clearly as possible. Other programs not meant for educa-

tional purposes tend to be cluttered with a lot of “whistles and bells” that obscure the finer, more important details.

With this book, you will be able to grasp the essence of an idea, use it in the program listing that illustrates the idea, and then fit it into some programming schemes of your own.

You will find many ideas in this book, but little razzle-dazzle.

Be assured at this point that the book is not devoted exclusively to machine-language programming. Many people who feel the itch to go beyond BASIC are told—or at least get the impression—that the next step in their programming experience must be in the direction of machine-language programming.

That is not true. Growing up in this business of computer programming is an evolutionary process. Your knowledge ought to develop gradually and smoothly. Moving directly from basic BASIC to pure machine-language programming is hardly a gradual and smooth process. In fact, it is a terrible mistake to drop BASIC and move to machine-language programming if you’ve had little training or experience with it. The change in thinking and technique is too big and too abrupt.

No, machine-language programming is not the first topic offered in this book. It turns out that familiar old Integer BASIC can become exciting again, once you know more about the internal workings of the Apple II. You can access some very useful monitor routines from BASIC and do a lot of things that will tear down some of the usual programming limitations.

And that’s where this book starts.

Once you know more about the system from a BASIC viewpoint, you will be ready to begin some assembly- and machine-language programming. But even then it will be in combination with BASIC programming. The idea is to let you wade into the deeper waters of machine language, while keeping a tight grasp on a familiar BASIC handle.

Toward the last part of the book, you will finally get to deal with pure machine-language and assembly-language programming. By that time, though, you will be well grounded in the Apple’s internal workings and better prepared to write successful machine-language programs of your own.

In short, if you are getting a little tired of your Apple system, this book ought to serve as a shot of adrenalin.

**HOW TO GET THE MOST FROM THIS BOOK** This is not really a reference book, although it might have that general appearance in many places. The book represents a step-by-step process. As such, you will get more from it by working through it from beginning to end, as opposed to dipping in at some point that seems interesting to you at the moment.



Since much of the first part of the book deals with decimal-oriented BASIC procedures, all of the tables in that part show only decimal values and addresses. To be sure, decimal addressing can be cumbersome, but it is a boon to readers who do not feel comfortable with hexadecimal addressing in the early going.

As the discussions flow in the direction of assembly-language and machine-language programming, you will find an increasing number of references to hexadecimal notation. Appendix A will be especially valuable to readers who have never used hex before.

You might do well to look through the extensive set of appendices in the back of the book now. You will find a large number of useful tables that present critical values and addresses in both hexadecimal and decimal form.

This book is a guide—a self-teaching guide. You will get the most from it by attempting to apply the new ideas in your own fashion. It is really based on the old notion that you can keep a man alive for a day if you give him a fish, but you can keep him alive for a good many years if you give him some fishing equipment and show him how to use it.

You have all the equipment; here come the ideas.



# Displaying Text

The most conspicuous parts of your Apple computer system are the keyboard and video screen, or crt. They ought to be the most conspicuous parts because they are the primary links between the human user and the sophisticated, fast-acting internal workings of the system. **2**

Most of what goes into the system from the human operator enters via the keyboard, and most of what comes out of the system to the human operator is displayed on the crt. Unless you happen to be running a graphics program, the computer will be communicating with you in a text format, i.e., with symbols, code words, or messages.

It is thus fitting that this book begin with a discussion of the crt and keyboard.

**THE STANDARD TEXT FORMAT** The video scheme of the Apple is arranged in such a way that it can display up to 40 characters on each horizontal line and up to 24 lines of text on the screen. That figures out to a maximum of 960 characters that can appear on the screen at any given moment. Of course, it is possible to print fewer than 40 characters per line and use fewer than 24 lines of text—these are simply the maximum figures.

Fig. 2-1 shows the video screen blocked off into its 960 possible character locations. The numerals across indicate the character, or column, number for each line of text. The numerals along the side of the drawing indicate the line, or row, numbers.

Notice that the character spaces, the blocks assigned to each character location on the screen, are not exactly square. Rather, they are a bit taller than they are wide.

Then notice how I have labeled the columns and rows beginning with the numeral 0. There are indeed 40 columns, but they are labeled 0 through 39. And there are 24 rows, but they are labeled 0 through 23. This is because zero is considered to be a counting number in the world of computing. The character space in the extreme upper left-hand corner of the screen is thus described by saying it is at column 0, row 0. A character

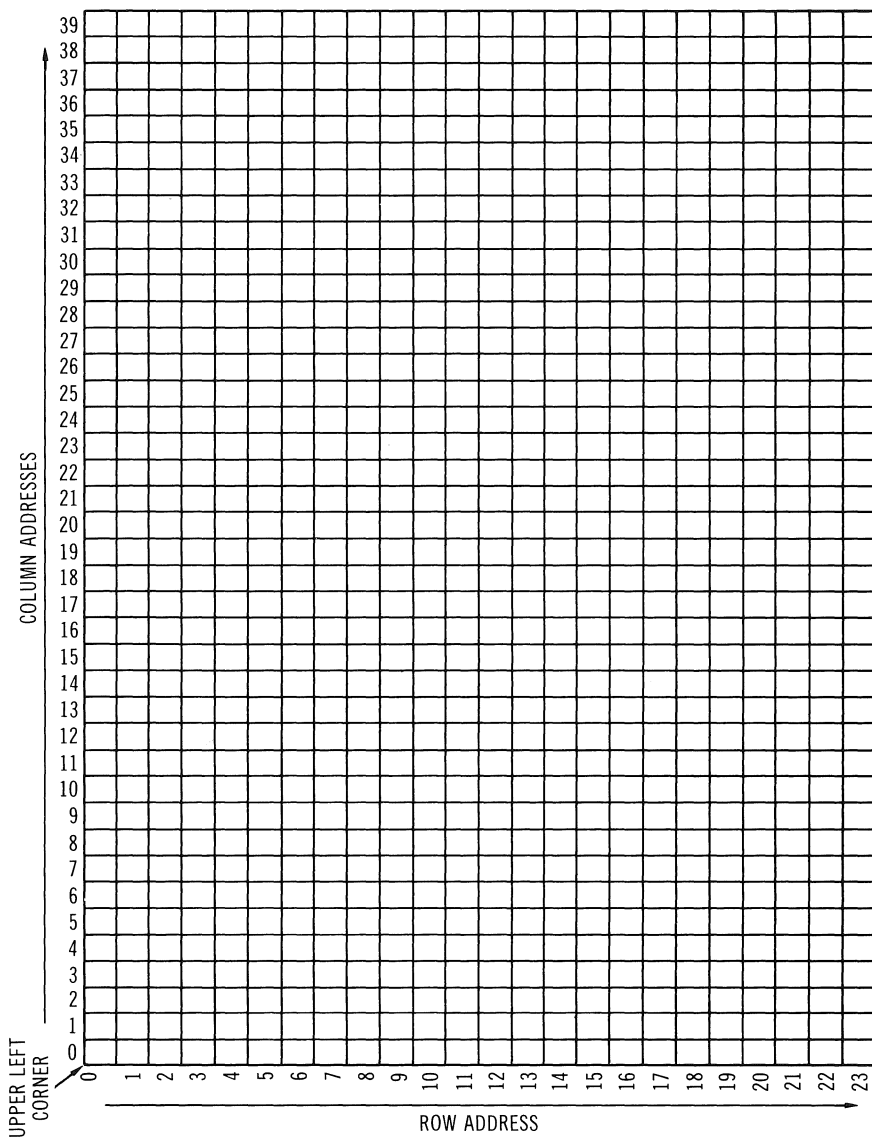


Fig. 2-1. Video screen character locations.



located in the extreme lower right-hand corner is described as being at column 39, row 23. A character location very near the middle of the screen has a coordinate address of column 19, row 11.

Virtually all video display operations use this column-and-row numbering format in one way or another. Normal video operations occur in a systematic fashion, i.e., from left to right. It is possible, however, to defeat the normal course of printing events and print a character at any column-and-row address you choose.

**ROLE OF THE CURSOR** In most kinds of video-printing operations, the *cursor* keeps track of where the next character is to be printed on the screen. The cursor is represented by a flashing rectangle of light in BASIC or monitor programming mode. The system knows where to display the cursor by referring to two specific memory locations that contain the column-and-row coordinates of the cursor.

Suppose that you are in Integer BASIC programming mode—the mode of operation signalled by a greater-than symbol at the beginning of a line. As you type in any arbitrary sequence of letters and numerals, you will see the flashing cursor symbol advancing along the line, always indicating exactly where the next character will be printed. If you fill a line with characters and continue typing more of them, the cursor automatically does a linefeed and carriage return. That is, it drops down one line and moves to the beginning of it. Screen printing then resumes from there. All through this operation, the Apple system is adjusting column-and-row addresses to reflect the position of the cursor.

The same sort of action occurs while the system is executing a programmed statement that calls for printing text on the screen. The only difference is that the cursor symbol is not displayed. There is simply no need for the computer to display the flashing cursor symbol when it is printing out preprogrammed characters. In other words, the cursor is meant to tell where the next character will be placed when someone is actually typing on the keyboard.

Suppose you have written a BASIC program that includes a PRINT “HELLO” statement. As the system executes that statement, it prints the H character at a given column-and-row location on the screen, advances the column address, prints the E, advances the column address, prints the first L, and so on.

The cursor-positioning mechanism works the same way whether you are operating in a programming or execution mode. The cursor, whether it is actually displayed or not, still indicates the position of the next character.

## CONTROLLING CURSOR POSITION WITH PRINT STATEMENTS

A BASIC PRINT statement causes the system to print either alphanumeric strings or numeric values on the screen. And unless the computer is directed to do otherwise, the information is PRINTed one character at a time, in a left-to-right fashion, with the “invisible” cursor leading the way.

**Simple PRINT Statements** A simple PRINT statement in BASIC is one that prints out a single PRINT element and then does an automatic linefeed and carriage return operation. Unless directed otherwise, the PRINTing begins in column 0 of the current line.

Suppose you execute this BASIC program:

---

```
10 FOR N=1 TO 4
20 PRINT "HELLO"
30 NEXT N
40 END
```

---

On RUNning that little program, you should see something like this on the screen:

```
HELLO
HELLO
HELLO
HELLO
```

The PRINT statement in program line 20 is executed four times in succession. Each time, the printing begins at column 0 and ends with an automatic linefeed and carriage return.

Unless directed otherwise, the system inserts a linefeed and carriage return at the end of every PRINT statement in a BASIC program.

This automatic feature operates whether the system is printing strings (as in the previous example) or numeric values. You will, for instance, see the same mechanism at work by running this program:

---

```
10 FOR N=1 TO 4
20 PRINT N
30 NEXT N
40 END
```

---

RUN that program, and you will see:

1  
2  
3  
4

Yes, the automatic linefeed and carriage return feature is at work here, too.

At the end of each simple PRINT statement, the system automatically sets the column address to 0 and increments the row address by 1. (Unless, of course, the next character is to appear below the last line on the screen. In that case the row address remains at 23 and everything else is scrolled up one line.)

**Suppressing Linefeed and Carriage Return** You can suppress the automatic linefeed and carriage return at the end of each PRINT operation by ending that operation with a semicolon (;). When you do that, the cursor picks up where it left off at the end of a previous PRINT statement. Try this example:

---

```
10 FOR N=1 TO 4
20 PRINT "HELLO";
30 NEXT N
40 END
```

---

The text on the screen should look like this:

HELLOHELLOHELLOHELLO

Sure enough, the automatic linefeed and carriage return is no longer automatic.

Concluding a PRINT statement in BASIC with a semicolon suppresses the automatic linefeed and carriage return.

The same idea applies to PRINTing numeric values:

---

```
10 FOR N=1 TO 4
20 PRINT N;
30 NEXT N
40 END
```

---

That one prints this sort of text:

1234

You can perform some useful and interesting text formatting by combining the automatic linefeed and carriage return feature with PRINT statements that suppress it. Suppose that you want to print out a  $4 \times 6$  array of X characters—6 lines of 4 Xs. Try this approach:

---

```
10 FOR ROW=1 TO 6
20 FOR COL=1 TO 4
30 PRINT "X";
40 NEXT COL
50 PRINT
60 NEXT ROW
70 END
```

---

The resulting text looks like this:

```
XXXX
XXXX
XXXX
XXXX
XXXX
XXXX
```

There are two different kinds of PRINT statements in that program. The one in line 30 prints a single X character on the screen and suppresses the linefeed and carriage return. The PRINT statement in line 50 really prints nothing onto the screen; but since it is not followed by a semicolon, the system will respond by doing a linefeed and carriage return. The result is that the “col” FOR-NEXT loop executes line 30 four times, creating a row of four Xs. The “row” FOR-NEXT loop executes the “col” FOR-NEXT loop and line 50 six times, creating six rows of Xs.

By way of a different example, consider that many extended BASICs include a STRING\$(n,c) function. This function prints a string of characters, represented by c, n times in succession. Integer BASIC does not include that function, but it's possible to simulate it this way:

The INPUT statements in line 20 let you specify the string character to be printed (C\$) and the number of them to be printed in succession (N). The subroutine beginning at line 100 simulates the action of a STRING\$ function. It features two kinds of PRINT statements: one ending with a semicolon for suppressing the linefeed and carriage return operation, and

---

---

```
10 DIM C$(64)
20 INPUT N: INPUT C$
30 GOSUB 100
40 GOTO 20
100 FOR X=1 TO N
110 PRINT C$;
120 NEXT X
130 PRINT
140 RETURN
```

---

one that does not conclude with a semicolon in order to execute the linefeed and carriage return at the end of the function.

**Setting Columns With Commas** Another way to suppress the linefeed and carriage return that normally occurs at the end of a PRINT statement is to end that statement with a comma (.). Recall that a PRINT statement ending with a semicolon allows the cursor to remain where it was at the end of the PRINT operation; the next PRINT operation picks up from there. A PRINT statement ending with a comma keeps the cursor on the same line, but forces it to advance to the beginning of the next column *field*.

The text screen can be divided into five equal fields of columns, each having 9 columns, or character spaces, in them. Those column fields begin at column addresses 0, 8, 16, 24 and 32. Try this demonstration program:

---

```
10 FOR N=1 TO 5
20 PRINT "HELLO",
30 NEXT N
40 END
```

---

RUN that little program, and you will find HELLO printed five times along the same line on the screen. Each HELLO begins at one of the well-defined column addresses for the five fields. Whenever the computer completes the task of printing the string HELLO, the comma in the program tells the cursor to advance to the right until it comes to the column address representing the beginning of the next field.

There are just 9 columns in each field, so it often happens that a PRINT statement will print a text that is longer than that; say, 12 characters long. When that is the case, the text extends well into the second field; but if it is ended with a comma, the cursor will begin printing the next line of text from the beginning of the third field—at column address 16. Comma suppression of the normal linefeed and carriage return is thus normally limited to printing operations involving fewer than 9 characters apiece.

## SETTING THE CURSOR POSITION WITH TAB STATEMENTS

PRINT statements in BASIC automatically adjust the cursor's horizontal and vertical position as necessary for the operation at hand. Ending PRINT statements with a semicolon or comma provides the programmer with some simple tools for controlling the automatic action of the cursor; but BASIC's TAB statements give the programmer full control over the cursor.

**The TAB Statement** The TAB statement in Integer BASIC lets the programmer set the column address of the cursor to any desired place along the current line of text. The TAB statement is an absolute addressing tool. That is to say, it makes no difference where the cursor was before TAB is used. The cursor is always moved to the column indicated by the TAB statement.

Unfortunately, an Apple TAB statement does not line up exactly with the column-numbering format we are using throughout this book. As described earlier, the columns are labeled 0 through 39. By contrast, the TAB statement uses labels 1 through 40. So doing a TAB 1 actually sets the cursor to column number 0 of the current line; and doing a TAB 40 sets it to the end.

For our purposes, then, the TAB statement has this syntax:

TAB c+1

where c is the column address for the cursor.

To see how this works, suppose that you want to print the string FOO at column address 8 and HELP at column address 24 of the same line. The following program does that for you:

---

```
10 TAB 9: PRINT "FOO";  
20 TAB 25: PRINT "HELP"  
30 END
```

---

The TAB 9 statement in line 10 sets the cursor to column address 8 of the current line. Then, the PRINT statement prints FOO and suppresses the linefeed and carriage return so that the HELP string appears on the same line. The TAB 25 statement in line 20 then sets the cursor to column address 24, and the PRINT statement prints HELP from that character location.

TAB addressing is absolute. It makes no difference where the cursor is located at the time the TAB statement is executed. You can demonstrate that by rewriting the previous example in such a way that the computer prints the right-hand string, HELP, first. Try this:

---

```
10 TAB 25: PRINT "HELP";
20 TAB 9: PRINT "FOO"
30 END
```

---

Line 10 calls for moving the cursor to column address 24 and printing the string `HELP` from that point. The semicolon following that `PRINT` statement prevents the system from doing a linefeed and carriage return. The `TAB 9` statement in program line 20 actually moves the cursor backwards to column address 8 before the `PRINT "FOO"` statement is executed.

The next program is an example of some `TAB` trickery. When entering the program, note that the `MOVE` string defined in line 10 is enclosed in spaces.

#### **Listing 2-1. TAB Special Effects.**

---

```
10 DIM M$(6):M$=" MOVE "
20 FOR COL=0 TO 33
30 TAB COL+1
40 PRINT M$;
50 FOR T=1 TO 100: NEXT T
60 NEXT COL
70 FOR COL=33 TO 0 STEP -1
80 TAB COL+1
90 PRINT M$;
100 FOR T=1 TO 100: NEXT T
110 NEXT COL
120 GOTO 20
```

---

Program lines 20 through 60 move the message string to the right one `TAB` location at a time. The time delay routine in line 50 slows down the action to an interesting pace. Lines 70 through 110 move that same message to the left.

Why are the `PRINT` statements in lines 40 and 90 terminated with a semicolon? Why is the `COL` variable advanced only to 33? Why is the `MOVE` message enclosed in spaces? If you find you cannot answer any one of these questions with confidence, edit the program to alter the items emphasized in the questions.

**The VTAB Statement** The `VTAB` statement is to row selection what `TAB` is to column selection. The Apple text screen format has 24 rows, or lines, that are often labeled with numerals 0 through 23. Integer

---

BASIC's VTAB statement is a row addressing function that uses numerals 1 through 24. The general syntax is:

VTAB r+1

where r is the row address.

The topmost row of characters is normally labeled row 0; but if you want to place the cursor there by means of a VTAB statement, a VTAB 1 would be appropriate. The following program illustrates its application:

---

```
5 CALL -936
10 FOR N=1 TO 4
20 VTAB N
30 PRINT "HELLO";
40 NEXT N
50 END
```

---

Beginning from the top line on the screen, the display looks something like this:

```
HELLO
  HELLO
    HELLO
      HELLO
```

No matter where the cursor might be when you RUN this program, the first HELLO appears on the first line of the screen. The fact that the PRINT statement in program line 30 ends with a semicolon means that each PRINT operation will *not* conclude with an automatic linefeed and carriage return. Thus the H character beginning a new message lines up against the 0 in the previous one, but on the next line down.

Try this variation of the same thing:

---

```
5 CALL -936
10 FOR N=1 TO 4
20 VTAB 2*N
30 PRINT "HELLO"
40 NEXT N
50 END
```

---



Beginning from line address 1, the text display looks like this:

```
HELLO  
HELLO  
HELLO  
HELLO
```

The successive VTAB values are 2, 4, 6 and 8. The actual row addresses are 1, 3, 5 and 7. Since the overall text presentation might be cluttered with characters you generated while entering the program, you might want to clear the screen first by doing an ESC @ and then a RUN.

Like the column addressing of the TAB statement, VTABs are absolute addresses. It makes no difference where the cursor might be located at the time the VTAB statement is executed. Try this:

---

```
10 CALL -936  
20 VTAB 1: PRINT "FOO"  
30 VTAB 11: PRINT "HELP"  
40 END
```

---

The CALL statement in line 10 clears the screen for you. Line 20 causes the string FOO to be printed on row address 0—the first line on the screen. Then line 30 prints HELP on row address 10.

To make sure you are convinced that the position of the cursor is not relevant when executing a VTAB, run this variation:

---

```
10 CALL -936  
20 VTAB 11: PRINT "HELP"  
30 VTAB 1: PRINT "FOO"  
40 END
```

---

The resulting text appears identical to the previous example, but the lower HELP message is printed before the FOO message is printed on row address 0.

**Using Combinations of TAB and VTAB** Programming, especially text formatting situations, can become a lot easier and more interesting when combining TAB and VTAB statements. Taken together, these two simple BASIC statements allow you to print a character at any one of the 960 text character locations on the screen. Generally speaking, you only have to make sure that you select the TAB and VTAB values so that the text to be printed does not overflow a line or crash into some previously printed text.

Here is a program that demonstrates how it is possible to place a single character just about anywhere on the screen:

---

```
10 CALL -936
20 PRINT "COLUMN ADDRESS (0-39)";
30 INPUT COL
40 PRINT "ROW ADDRESS (0-23)";
50 INPUT ROW
60 CALL -936
70 TAB COL+1: VTAB ROW+1
80 PRINT "X";
90 GOTO 90
```

---

The program requests the column and row addresses for the cursor. Respond by entering a value in the designated range in each case. Once you've entered those cursor addresses, the program clears the screen and prints an X in that location. The program ends by looping to itself at program line 90, so you must do a CTRL C and another RUN to try a different set of cursor coordinates,

The only problem with the whole idea is that it doesn't print the X properly whenever you specify the last character location in the lower right-hand corner of the screen—column 39, row 23. The program will indeed print the X at that character location, but then the system does its normal task of advancing the cursor to the next character location. In this particular case, that means the entire display will scroll upward one line, and the X will not remain in the designated position. None of the other 959 possible character locations will cause that undesirable scrolling effect.

Here is a short program that prints stars (asterisks) at randomly selected character locations on the screen. Notice that it avoids printing asterisks on the bottom line. Why? Because printing an asterisk at column 39 in row 23 would mess up the overall effect by scrolling it upward one line.

---

```
10 CALL -936
20 TAB RND (39)+1
30 VTAB RND (22)+1
40 PRINT "*";
50 GOTO 20
```

---

The next program is a variation of the one just described. It still prints asterisk characters at randomly selected places on the screen, but this time it erases each one after displaying it for a short interval of time. The overall effect is that of stars twinkling here and there.

---

---

```
10 CALL -936
20 COL= RND (39)+1:ROW= RND (23)+1
30 TAB COL: VTAB ROW
40 PRINT "*";
50 FOR T=1 TO 10: NEXT T
60 TAB COL
70 PRINT " ";
80 GOTO 20
```

---

Program line 20 selects random values that are appropriate for the TAB and VTAB statements in line 30. Line 40 then prints the asterisk character at the selected screen position, and line 50 does a short time delay.

Lines 60 and 70 work together to delete the asterisk at the end of the time delay interval. The general idea is to plot a space character (program line 70) over the asterisk; but without the TAB COL instruction in line 60, the space would always be printed one column to the right of the asterisk.

Bear in mind that any PRINT statement, including those appended with a semicolon to suppress the normal linefeed and carriage return, advances the cursor to the next column, or character space. So, if the program happens to print the asterisk in row address 5 of some line, the cursor will end up at row address 6. The TAB COL statement in line 60 of the program “backs up” the cursor, placing it at the address of the asterisk character and forcing the space of line 70 to print over it.

Of course the cursor symbol is not displayed during the execution of a program, so the program isn’t disturbed by that flashing symbol. The mechanisms for positioning the cursor are nevertheless at work here.

**Simulating a PRINT @ Statement** A lot of BASICs include a PRINT @ statement that, in a fashion, performs the task of the combined TAB and VTAB statements described in the previous section.

The Apple text screen is divided into 960 different character locations, and we have been addressing those locations by means of some column-and-row coordinates—that is, by designating both a row address and a column address. A PRINT @ statement accomplishes the same thing, but it considers the addresses for the 960 character locations in a different manner.

A PRINT @ statement designates a single numeric value for each character location on the screen. There are 960 character locations in the Apple text format, so the range of PRINT @ addresses is 0 through 959. The PRINT @ addresses begin with 0 in the extreme upper left-hand corner of the screen, and progress in a column-by-column and row-by-row order until they reach address 959 in the extreme lower right-hand corner. PRINT @ address 39 is at the end of the first row, PRINT @ address 40 is at the beginning of the second row, and so on.

The usual syntax for a PRINT @ statement is:

PRINT @ x,c

where,

x is the PRINT @ address (0-959)

c is the character to be printed at the designated address.

Here is a program that uses Integer BASIC's TAB and VTAB statements to simulate and demonstrate the PRINT @ operation.

---

**Listing 2-2 PRINT @ Simulation.**

---

```
10 CALL -936
20 PRINT "WHAT PRINT @ VALUE (0 TO 959)";
30 INPUT X
40 IF X>=0 AND X<=959 THEN 70
50 PRINT "** RANGE ERR"
60 CALL -198: GOTO 20
70 PRINT "WHAT CHARACTER ";
80 INPUT C$
90 CALL -936
100 GOSUB 200
110 GOTO 110
200 ROW=X/40
210 COL=X MOD 40
220 TAB COL+1: VTAB ROW+1
230 PRINT C$;
240 RETURN
```

---

Program line 20 requests a PRINT @ address in the range of 0 through 959. If you input a value outside that range, lines 50 and 60 print the familiar \*\* RANGE ERR message and beep the loudspeaker. After inputting a valid PRINT @ address, lines 70 and 80 allow you to designate a string character to be printed at that location.

The subroutine beginning at line 200 represents the real purpose of the program: to demonstrate TAB and VTAB cursor positioning. That subroutine calculates the individual ROW and COL addresses, line 220 sets up the TAB and VTAB operations, and line 230 prints the designated string character at that position on the screen. It's a matter of converting a single-number address that is in the range of 0 through 959, into a pair of TAB and VTAB values that are consistent with the operation of Apple Integer BASIC.

The program, incidentally, loops to itself at line 110 after printing the character. So you have to enter a CTRL C and RUN to try the PRINT @ demonstration again.

---

**WORKING WITH THE CURSOR-POSITION REGISTERS** There are two memory locations in Apple RAM that spell out exactly where the cursor is located at any given moment and under any operating mode. The numeric values stored in those locations follow the same column-and-row addressing format used throughout this chapter.

The Apple literature cites those two RAM addresses as CH and CV:

CH at RAM address 36—this register specifies the cursor's column location (0–39).

CV at RAM address 37—this register specifies the cursor's row location (0–23).

PEEKing and POKEing into those two addresses gives you total control over the positioning of the print cursor.

**PEEKing Into CH and CV** Memory address 36 always holds the cursor's current column address; so including a BASIC statement such as

**PRINT PEEK(36)**

in your program prints out that value for you. But if you try that statement while in immediate execution mode, the system always responds by printing a 0. Why would CH be set to 0 in such a case? Because statements in immediate mode are executed only after you strike the RETURN key. Striking the RETURN key amounts to doing a linefeed and carriage return, and that always forces the column address to 0. Hence a simple PRINT PEEK(36) statement is virtually useless in the immediate execution mode of operation.

Build a PRINT PEEK(36) into a program, though, and you can see the column address register, CH, at work. Try this little demonstration:

---

```
10 FOR N=1 TO 4
20 PRINT "*";
30 NEXT N
40 PRINT PEEK (36)
50 END
```

---

Running that program turns up a text display that looks something like this:

```
****4
```

The FOR-NEXT statement prints asterisks in column addresses 0, 1, 2, and 3. The system advances the cursor to the next column, and then the

PRINT PEEK(36) statement in line 40 prints out the current cursor column address—column 4 in this case.

Here is a program that lets you experiment with the notion of PEEKing into CH:

---

```
10 PRINT "INPUT A COLUMN ADDRESS (0-39)";
20 INPUT COL
30 FOR N=0 TO COL: PRINT "X";: NEXT N
40 HPOS= PEEK (36)
50 PRINT : PRINT HPOS
60 GOTO 10
```

---

The program first requests a column address for the current line of text. Answer it by typing any integer value between 0 and 39. After doing that, notice that the program prints a line of X characters from column address 0 of the current line to the address you specified. Program line 40 is the one of special interest here: it PEEKs into CH and assigns the current column address to variable HPOS. Line 50 then prints that value on the screen for you.

So if you respond to the input request with a column address of 20, the program prints 21 X characters on the next line and then prints the numeral 21 on the line under that one. The program prints 21 X characters in this particular case because it is filling in column addresses 0 through 20, which consist of 21 locations. It prints a value of 21 for HPOS because the printing operation ended at column address 20, after which the cursor automatically advanced to the next column—column address 21.

Incidentally, if you omit the semicolon at the end of the PRINT statement in line 30, the program will always print a value of 0 for HPOS. Why?

Using the same line of thinking, you can assure yourself that CV (RAM address 37) always carries the cursor's current row address. You can take a look at the content of CV at any time by doing something such as:

### PRINT PEEK(37)

Doing that, you will see a number anywhere between 0 and 23, depending on the current row address.

Here is the program for PEEKing into the CV register:

---

```
10 PRINT "INPUT A ROW ADDRESS (0-23)";
20 INPUT ROW
30 CALL -936
40 FOR N=0 TO ROW: PRINT "X": NEXT N
50 VPOS= PEEK (37)
60 PRINT VPOS
70 GOTO 10
```

---

This one asks you to designate a cursor row address. The program then clears the screen and prints X characters from the beginning of row number 0 to the row address you specified. Line 50 then PEEKs into CV and assigns the value to VPOS. Line 60 prints that value for you.

How can you find out where the cursor is located at any given moment? Simply PEEK into address 36 to find the column address and PEEK into address 37 to find the row address.

Some versions of BASIC include statements that PEEK into those cursor-address locations for you. The syntax generally includes the use of a dummy variable, so they take on this sort of form:

```
LET COL=HPOS(X)
LET ROW=VPOS(Y)
```

where X and Y are dummy variables. (They are required for proper execution of the statement, but have no real significance beyond that.)

The hypothetical HPOS(X) function does the job of our PEEK(36), and the VPOS(Y) does the job of our PEEK(37).

Would you like to come up with a PRINT @ address for the current cursor location? One that expresses the cursor location as a single-number value between 0 and 959? This sort of statement will do the job:

```
PAT=PEEK(36)+PEEK(37)*40
```

The PEEK(36) picks up the content of CH to get the cursor's column address, and PEEK(37) looks into CV to get the row address. Multiplying the content of CV by 40 and summing the result with CH generates a PRINT @ sort of number that is assigned to variable PAT. It will always be a number between 0 and 959, and it will indicate the cursor's position in a form described for PRINT @ statements in "simulating a PRINT @ statement" on page 29.

**POKEing Into CH and CV** The Apple system refers to RAM addresses CH and CV whenever it is necessary to use them for the sake of knowing where the next character is to be printed on the screen. You found in the previous section that you can find those addresses by PEEKing into CH and CV. Now you will see that you can actually control the position of the cursor by POKEing numbers—column and row addresses—into CH and CV. In fact you will find that POKEing into CH and CV lets you perform described for PRINT @ statements in "Simulating a PRINT @ Statement" on page 29.

This program demonstrates the feasibility of POKEing values into CH in order to begin printing a message at a designated column address:

---

```
10 CALL -936
20 PRINT "WHAT COLUMN STARTING ADDRESS (0-39)";
30 INPUT COL
40 POKE 36,COL
50 PRINT "HELP"
60 GOTO 20
```

---

Enter and run this program, responding to the request for a column address with an integer value between 0 and 39. The program responds by printing the HELP message beginning at the column address you specify. How is that done? Line 40 in the program POKes your COL value into CH. The system then uses that value in CH to fix the starting column for the message that is spelled out in line 50.

Try several different COL values, and convince yourself that the HELP message in each case begins at the address value you specify.

Every time the system is called upon to print a text character, it refers to the content of CH to determine the column address of the character. After printing the character, the system increments the value in CH to place the cursor at the next column address.

POKEing values into CH and getting the desired result is a rather straightforward procedure. Unfortunately, the Apple system isn't set up to work with POKes into CV in such a straightforward manner. The Apple system ignores the row-address value in CV unless it is really needed; it does not refer to that value after every character-printing operation.

The Apple system refers to the content of CV only after it sees a linefeed and carriage return or notices that a printing operation is moving the cursor beyond the right end of the current row. That makes the matter of setting the row address by POKeing into CV (RAM address 37) a less-than-ideal formatting procedure.

For example, you might think this would be a good program for demonstrating the notion of setting the row address by POKeing into CV:

---

```
10 CALL -936
20 PRINT "WHAT ROW ADDRESS (0-23)";
30 INPUT ROW
40 POKE 37,ROW
50 PRINT "HELP"
60 GOTO 20
```

---

When you enter a value for variable ROW, program line 40 POKes it into CV. You might think that the system uses that value to determine the

---



row address for printing the HELP message. But it doesn't! No matter what value you input for ROW, the next format looks like this:

```
WHAT ROW ADDRESS (0-23)?  
HELP
```

The HELP message appears on the row directly following the row-address request.

Notice, however, that the next request for an address appears at the row address you specified. The system does not refer to your specified row address until after it prints the HELP message. I'll have to confess that it took me some time to figure out what was going wrong when I tried this sort of POKEing into CV.

The principle of the thing is that the Apple system, in the interests of operating speed, does not refer to the content of CV until it is time to set up a new line of text. It refers to CH after every character-printing operation, but not CV. In this particular example, the system refers to the content of CV only after printing the HELP message. Why does it look into CV then? Because the PRINT "HELP" statement ends with an automatic linefeed and carriage return. The process for doing a linefeed and carriage return is one that makes the Apple check the value in CV.

So to set up the HELP messages at the desired ROW address, we have to do something that makes the Apple refer to CV *after* POKE 27, ROW, but before PRINT "HELP". Try adding the following line to the program just described:

```
45 CALL -926 : CALL -998
```

Now you will find the program runs as expected.

The CALL statements in program line 45 do a linefeed and carriage return followed by an upward linefeed. Both CALLs affect CV and force the Apple to refer to it; the first CALL forces the cursor down a line, and the second CALL puts it back. Everything works out nicely.

Of course you can combine these programs to take full control over the horizontal and vertical positioning of the cursor:

---

```
10 CALL -936  
20 PRINT "WHAT COLUMN ADDRESS (0-39)";  
30 INPUT COL  
40 PRINT "WHAT ROW ADDRESS (0-23)";  
50 INPUT ROW  
60 POKE 37,ROW  
70 CALL -926: CALL -998  
80 POKE 36,COL  
90 PRINT "HELP"  
100 GOTO 20
```

---

Why do you suppose it is necessary to POKE the value of ROW into CH and execute the two CALL statements (lines 60 and 70) *before* setting COL into CH (line 80)? Hint: CALL -926 (a linefeed and carriage return) is the key to the answer.

**MORE CURSOR-RELATED OPERATIONS** All of the cursor-related operations described thus far deal with the cursor and its role in printing characters on the screen. There is a family of other cursor-related operations that perform other, equally important tasks. Those tasks include moving the cursor up, down, and to the left or right from its present position; and clearing selected portions of the text display.

Many of these special cursor-related tasks can be executed by taking advantage of some simple CALL routines to the Apple monitor. Others cannot.

**Homing the Cursor** *Home*, as far as the cursor is concerned, is that character position in the extreme upper left-hand corner of the text screen. Homing the cursor is a process that places it there without disturbing any other text on the screen.

A couple of previous discussions in this chapter offer some techniques for homing the cursor from any other location.

First, you can TAB and VTAB it to home with statements such as:

**TAB 1: VTAB 1**

When using full-screen text, TAB and VTAB values of 1 represent *home* for the cursor. Here is a demonstration program that uses this particular homing technique:

---

```
10 DIM M$(16)
20 TAB 1: VTAB 1
30 INPUT M$
40 GOTO 20
```

---

Program line 20 homes the cursor just prior to the execution of the INPUT statement in line 30. So every INPUT routine begins from the upper left-hand corner of the screen.

Second, you can POKE zeros into CH and CV:

---

```
10 DIM M$(16)
20 POKE 37,0: CALL -926: CALL -998
30 POKE 36,0
40 INPUT M$
50 GOTO 20
```

---

Lines 20 and 30 work together to home the cursor just before the program executes the INPUT statement in line 40. The overall operation is identical to the TAB/VTAB version cited before.

Unfortunately, the Apple monitor does not include an entry point that simply homes the cursor by means of a CALL statement in BASIC. A short machine-language routine will let you do the job in a custom fashion; but that is a topic that we will discuss much later in this book.

**Home the Cursor and Clear the Screen** The most likely reason that there is no simple CALL statement for just homing the cursor is that most programmers want to link homing the cursor with clearing the entire screen. Such a two-part operation should be called “home the cursor and clear the screen,” but it generally goes by the simple name of HOME. HOME, in fact, is the designation assigned to this two-part operation in most of the Apple literature.

Most Apple users, particularly those who use Integer BASIC, are well acquainted with the HOME operation:

#### CALL -936

Executing that statement, either in the programming mode or the immediate execution mode, sends the cursor to its home position and clears the entire text field.

HOME (-936) is the entry point for operations that both home the cursor and clear the text screen.

If you have been entering and running the demonstration programs offered so far in this chapter, you have been working with the CALL -936 instruction. So there is no need for citing any further examples.

You can also call up that home-and-clear operation from BASIC command mode or from the monitor by doing an ESC-@ key function. Strike the ESC key and then type an @. As you can see, that too homes the cursor and clears the screen.

**Clear to End of Line** Sometimes it is helpful to clear a line of text from the current cursor position to the end of the current line of text. Try this demonstration program:

---

```
10 DIM M$(39)
20 TAB 1: VTAB 1
30 INPUT M$
40 CALL -868
50 GOTO 20
```

---

When you get the program loaded and running, respond to the INPUT statement by typing in an arbitrary string of characters—say, 16 or so. Do a RETURN and input another string of characters that is somewhat shorter than the first. When you do the RETURN for that second entry, you will see the characters at the end of the previous entry cleared from the screen.

Line 20 in the program homes the cursor without clearing anything from the screen, and line 30 lets you type in a string of characters from that point. The statement in line 40 is the one that is important to the present discussion. It CALLs an entry point in the Apple monitor that clears any text that resides in the space from the current cursor position to the end of the line. In the Apple literature, entry address -868 is called CLREOL—clear to end of line.

CLREOL (-868) is the entry point for operations that clear text from the cursor's current column address to the end of the current row, or line.

You can also execute CLREOL from the keyboard by doing an ESC-E—that is, by pressing ESC followed by E.

**Clear to End of Page** Just as CALLing CLREOL clears a line of text from the current cursor position to the end of the current line, CALLing CLREOP clears text from the current cursor position to the *end of the page*.

Here is a CLREOP demonstration program:

---

```
10 FOR N=0 TO 39: FOR M=0 TO 23
20 PRINT "*";
30 NEXT M: NEXT N
40 TAB 1: VTAB 1
50 PRINT "WHAT COLUMN NUMBER (0-39)";
60 INPUT COL
70 PRINT "WHAT ROW NUMBER (0-23)";
80 INPUT ROW
90 TAB COL+1: VTAB ROW+1
100 CALL -958
110 END
```

---

This one first fills the entire screen with asterisk characters (program lines 10 through 30). Line 40 homes the cursor without erasing any of the asterisks along the way, and lines 50 through 80 let you enter cursor-positioning addresses. Line 90 forces the cursor to the prescribed position

on the screen, and line 100 CALLs CLREOP at address -958 to clear the screen from that cursor position to the end of the page.

CLREOP (-958) is the entry point for operations that clear text from the current cursor location to the end of the page.

You can also execute CLREOP from the keyboard by doing an ESC-F command—that is, by pressing ESC followed by F.

Recall that CALLing HOME at address -936 both homes the cursor and clears the entire text screen. It is possible to do the same thing this way:

---

```
10 TAB 1: VTAB 1
20 CALL -958
30 END
```

---

Line 10 homes the cursor and line 20 does a CLREOP to clear the screen all the way from the home position to the end of the screen.

**Advance the Cursor** Another kind of monitor routine, called ADVANCE, begins at address -1036. Its function is to advance the cursor one character location to the right. ADVANCE differs from doing a SPACE keystroke inasmuch as ADVANCE does not erase characters as it moves the cursor position. What's more, ADVANCE will do an automatic linefeed and carriage return when executed at the end of a line, and it will scroll the entire screen upward when the cursor is at the last character position on the last line.

So including an ADVANCE operation, CALL -1036 in a FOR-NEXT loop lets you position the cursor in a left-to-right fashion without erasing any of the characters that might be in the cursor's path.

ADVANCE (-1036) is the entry point for an operation that moves the cursor one column address to the right without erasing the current character.

An ESC-A combination of keystrokes from the keyboard calls up the ADVANCE routine from the BASIC command mode or the monitor. Striking the right-arrow key does essentially the same thing.

You might also take note of the fact that a series of successive ADVANCE operations can do the same job as a TAB statement. There is one

big difference, however. TAB statements use *absolute* column addressing—that is, the value assigned to the TAB statement refers to the column position as reckoned from the left-hand edge of the text field. So if the cursor happens to be at column address 6 and you want to move it ten spaces to the right, you have to take into account the cursor's current position and specify a TAB 17. But doing ten ADVANCE operations in succession will move the cursor ten column locations to the right from its current location. That is an example of *relative* cursor addressing—addressing that is relative to the current cursor position.

**Backspace the Cursor** The BS routine in the monitor begins at address -1008, and its function is to move the cursor one space to the left. Doing a succession of these backspace operations moves the cursor in a right-to-left fashion without erasing any characters that might be in its way.

BS (-1008) is the entry point for an operation that moves the cursor one column address to the left without erasing the current character.

You have most likely used the routine a number of different times by striking the left-arrow key. You can do the same thing by pressing the key-stroke combination, ESC-B.

Like ADVANCE, BS is an example of relative cursor addressing. Inserting a CALL -1008 into a FOR-NEXT loop backspaces the cursor a number of locations relative to its starting position.

**Downward Linefeed** You can always do a downward linefeed by striking the ESC key, followed by striking the C key. That moves the cursor down one line—straight down. A RETURN keystroke also moves the cursor down one line, but to the beginning of that line.

You can include a downward linefeed operation in a BASIC program by executing a CALL -922. That address marks the beginning of the monitor's LF routine.

LF (-922) is the entry point for an operation that moves the cursor down one line, or row, without erasing any characters in its path.

The LF operation represents a relative-addressing version of BASIC's VTAB statement.

**Upward Linefeed** The monitor's UP routine, beginning at address -998, moves the cursor straight up one line. One or more CALL -998

statements in a BASIC program effectively move the cursor upward without affecting the text.

UP (-998) is the entry point for an operation that moves the cursor up one line, or row, without erasing any characters in its path.

An ESC-D operation lets you do the same thing from the keyboard.





# Alternative Text Formats

The Apple offers some alternatives to the standard text formats described in Chapter 2. The alternatives include inverse and flashing characters, and custom text windows. It is certainly possible to run some useful and sophisticated text routines without using any of these alternative formats, but their availability offers you a chance to turn a good program into a great one.

3

**ALTERNATIVE CHARACTER FORMATS** Unless directed otherwise, the Apple system prints text characters in a white-on-black format. An initialization routine that is built into the Apple monitor automatically sets up the standard white-on-black format whenever you turn on the system or do a RESET. You can, however, set up the text so that *all characters* appear in the inverse, black-on-white, format. You can also set up the text so that the *@ symbol* and *all letters* appear in the flashing format.

The key to setting up these formats is RAM location 50, otherwise known as INVFLG to the Apple operating system. POKEing appropriate values into INVFLG makes the text display normal, inverse, or flashing (see Table 3-1).

Notice from the table that the normal and inverse text formats apply to all printed characters, while the flashing text format does not. This is a peculiarity of the Apple that can cause some frustration if you are not aware of it.

The flashing text mode applies only to the @ symbol and the alphabet, but all characters may be printed in the inverse format.

You can set the text modes directly from the keyboard in the command mode of operation or you can write the appropriate POKES into a BASIC program.

**Table 3-1. Text Formats**

<b>Text Format</b>	<b>Content of INVFLG (RAM address 50)</b>	<b>Example</b>	<b>Notes</b>
NORMAL	255	POKE 50,255	All characters are printed in a normal, white-on-black format.
INVERSE	63	POKE 50,63	All characters are printed in an inverse, black-on-white format.
FLASHING	127	POKE 50,127	The @ symbol and all letters are printed in a flashing format; all other symbols and numerals are printed in the INVERSE format.

**Setting Text Formats From the Keyboard**      The following series of short experiments demonstrates how to set text formats from the keyboard.

1. Enter this program:

---

```

10 CALL -936
20 FOR N=1 TO 12
30 PRINT "X";
40 NEXT N
50 END

```

---

Line 10 homes the cursor and clears the screen. Lines 20 to 40 print 12 X characters in succession on the same row.

2. From the command mode, enter:

```

POKE 50,63
RUN

```

The POKE statement sets up the inverse text format so that when you run the program, the 12 X characters appear black-on-white.

3. From the command mode, enter:

```

POKE 50,127
RUN

```

---

The POKE statement sets up the flashing text format, so you should see the X characters flash.

4. From the command mode, enter:

```
POKE 50,255  
RUN
```

That POKE statement returns the system to the normal, white-on-black text format, so the program ought to print the 12 X characters in that fashion.

Steps 2, 3, and 4 demonstrate that you can set the format from the keyboard prior to running a text-printing program.

5. From the command mode, enter:

```
POKE 50,63  
RUN
```

The 12 X characters should appear in the inverse form.

6. Now, perform a RESET, a CTRL-C, and enter:

```
RUN
```

The X characters should now appear in the normal text format.

Steps 5 and 6 show that you can return to the normal text format by resetting the system. Thus there are two ways to get back to the normal text mode: by doing a POKE 50,255 or a RESET followed by a CTRL-C.

7. Rewrite line 30 in the program in Step 1 to read:

```
30 PRINT "3";
```

That simply replaces the 12 X characters with numeral 3 characters.

8. Repeat Steps 2, 3, and 4.

The numerals appear as one might expect them to appear in Steps 2 and 4—inverse and normal, respectively. But they do not appear in the flashing-text mode following the POKE 50,127 as prescribed in Step 3. The point of the demonstration is to show that numerals cannot be made to flash.

**Setting Text Formats Within a Program**     The previous set of experiments showed how you can set the text format from the keyboard prior to running a print-oriented program. The next program shows how you can set the text modes within a program.

---

```
10 CALL -936
20 PRINT "WHAT CHARACTER?";
30 INPUT C$
40 CALL -936
50 POKE 50,63: GOSUB 100
60 POKE 50,127: GOSUB 100
70 POKE 50,255: GOSUB 100
80 PRINT : PRINT
90 GOTO 20
100 FOR N=1 TO 12
110 PRINT C$;
120 NEXT N
130 PRINT
140 RETURN
```

---

Enter the program and run it. Respond to the WHAT CHARACTER request by entering a single letter, numeral, or punctuation symbol. The program will respond by printing three rows of 12 characters apiece, each row having a different text format: inverse, flashing, and normal. Remember, though, that characters other than @ and a letter of the alphabet will not flash.

Here is an analysis of that program:

Line 10 homes the cursor and clears the screen.

Lines 20 and 30 request a character and input it as string variable C\$.

Line 40 homes and clears again.

Line 50 sets the inverse text format and calls subroutine 100 to print 12 characters in a row.

Line 60 sets the flashing text format and calls subroutine 100 to print 12 characters in a row.

Line 70 sets the normal text format and calls subroutine 100 to print 12 characters in a row.

Lines 80 and 90 skip two lines on the screen and then loop back to program line 20 to request another character.

Lines 100 through 140 constitute a subroutine for printing 12 C\$ characters in a row. The text format is set just prior to calling this subroutine.

The program clearly demonstrates you can set the text format, and indeed change it, during the execution of a program.

---

## Mixing Text Formats

One of the most compelling reasons for using inverse or flashing text formats is to make important segments of printed text stand out clearly to get the user's attention. Mixing inverse or flashing text with normal text is a simple matter of switching text formats at critical points within a message-printing operation. Here is an example:

---

```
10 CALL -936
20 POKE 50,255
30 PRINT "ENTER A NUMBER ";
40 POKE 50,63
50 PRINT "BETWEEN 0 AND 9 ";
60 POKE 50,255
70 INPUT N
80 IF N>=0 AND N<=9 THEN 100
90 PRINT : POKE 50,127: GOTO 50
100 CALL -936
110 PRINT "YOUR NUMBER WAS ";
120 POKE 50,63
130 PRINT N
140 PRINT : PRINT
150 GOTO 20
```

---

Enter and run this program, and you will see the request: ENTER A NUMBER BETWEEN 0 AND 9. The first part of the text, ENTER A NUMBER, is written in the normal, white-on-black form. The last part, however, appears as inverse text. The idea is to emphasize BETWEEN 0 AND 9.

If you respond to the request by entering some integer between 0 and 9, the program prints YOUR NUMBER WAS followed by the numeral you specified. YOUR NUMBER WAS is printed in normal format, but the numeral appears in inverse format to make it stand out from the rest of the text. After doing that, the program loops back to request another number.

If you respond with an integer value that is outside the range of 0—9, the program prints BETWEEN 0 AND 9 in flashing format. The idea is to emphasize that you've made an error. In that error message, the words BETWEEN and AND are flashing; numerals 0 and 9 are in inverse form since numerals cannot be made to flash.

Bear in mind that the purpose of this program is to show how you can insert statements that alter the text formats to emphasize ideas that are important to the user. Here is a brief analysis of the operations:

Line 10 homes the cursor and clears the screen.

Lines 20 and 30 set the normal text format and print ENTER A NUMBER without the usual carriage return.

Lines 40 and 50 set the inverse text format and complete the request message by printing BETWEEN 0 AND 9.

---

Lines 60 and 70 set the normal text format and input the numeral as variable N.

Line 80 tests the value of N. If N is within the requested range of values, jump to program line 100; otherwise it executes the program from line 90.

Line 90 skips a line on the screen, sets the flashing text format, and loops back to program line 50 to print BETWEEN 0 AND 9 as an error message.

Line 100 homes the cursor and clears the screen.

Line 110 prints YOUR NUMBER WAS in the normal text format set in line 60.

Lines 120 and 130 set the inverse text format and print the value of variable N.

Lines 140 and 150 skip two lines on the screen and then loop back to program line 20 to request another number.

Notice especially how one can use the same message statement, BETWEEN 0 AND 9 in line 50, to serve two different purposes. Printed in inverse format, it simply emphasizes that the specified number should be in that range. Printing the same message in flashing format makes it serve as an eye-catching error message.

Mixing text formats within an operating program can transform an otherwise humdrum program into something a lot more interesting for the user. An example is the typical two-player high-low guessing game shown in Listing 3-1. Enter the program, run it, and play with it for a while. Take note of how it uses different text formats to add interest.

A step-by-step analysis of how the game works is left to you. It is more important at this time to point out the ideas behind changing the text modes.

First notice program lines 1000, 2000, and 3000. Each does nothing more than set a particular text format and return to the statement that calls it as a subroutine. Line 1000 sets the normal text format, 2000 sets the inverse format, and 3000 sets the flashing format. It isn't necessary to write those text-setting POKE statements as subroutines, but I've done it here to make them easy to use in other programs.

Thus, having the GOSUB 1000 statement in the main part of the program is the same as having a POKE 50,255 there, because they both set the normal text format for any PRINT statements that follow. (See program lines 50, 190, 210, 360, and 380.)

Along those same lines, GOSUB 2000 effectively sets up the inverse text format, and GOSUB 3000 sets up the flashing format.

You should be able to get a good appreciation of how the mixed-text scheme works if you run the program and follow the listing as you go along.

### Listing 3-1. High-Low Guessing Game.

---

```
10 DIM P1$(16),P2$(16),PN$(16)
15 CALL -936
20 TAB 15: VTAB 10
30 GOSUB 2000: PRINT "HIGH-LOW GAME"
40 VTAB 16
50 GOSUB 1000: PRINT "STRIKE ";
60 GOSUB 2000: PRINT "RETURN";
70 GOSUB 1000: PRINT " TO BEGIN ...";
80 INPUT S$
90 CALL -936
100 PRINT "FIRST PLAYER'S NAME ... ";
110 INPUT P1$
120 PRINT : PRINT "SECOND PLAYER'S NAME ... ";
130 INPUT P2$
140 PN=1
150 CALL -936:X= RND (99)+1:S=0
160 IF PN=1 THEN PN$=P1$
170 IF PN=2 THEN PN$=P2$
180 GOSUB 2000: PRINT PN$;
190 GOSUB 1000: PRINT " I AM THINKING OF A NUMBER"
200 PRINT "BETWEEN 0 AND 100"
210 GOSUB 1000
220 PRINT : PRINT "WHAT IS YOUR GUESS?"
230 INPUT G:S=S+1
240 IF G>=0 AND G<=100 THEN 260
250 PRINT : GOSUB 3000: GOTO 200
260 CALL -936
270 PRINT PN$;"'S GUESS NO. ";S;":"
280 PRINT : GOSUB 2000
290 PRINT G;" IS ";
300 IF G>X THEN 330
310 IF G=X THEN 340
320 GOSUB 2000: PRINT "TOO LOW": GOTO 210
330 GOSUB 2000: PRINT "TOO HIGH": GOTO 210
340 GOSUB 3000: PRINT "RIGHT ON"
350 PRINT : PRINT
360 GOSUB 1000: PRINT "STRIKE ";
370 GOSUB 2000: PRINT "RETURN";
380 GOSUB 1000: PRINT " TO CONTINUE THE GAME ...";
390 INPUT S$
400 IF PN=1 THEN 420
410 PN=1: GOTO 150
420 PN=2: GOTO 150
1000 POKE 50,255: RETURN
2000 POKE 50,63: RETURN
3000 POKE 50,127: RETURN
```

**ALTERNATIVE PRINT WINDOWS** The Apple text environment is organized into 24 rows, or lines, having 40-character columns in each. At least that is the normal text environment. It represents the largest possible print window—960 character locations as outlined in Fig. 2-1. If you type for a while, you will find that you can print up to 40 characters on as many as 24 different lines. Attempt to print on an imaginary 25th line, and you will see the scrolling effect.

Recall that starting up the computer or doing a RESET operation automatically sets the system for printing characters in their normal, white-on-black form. The same sort of thing happens to the print window: turn on the system or do a RESET, and the system automatically sets up the 40 column, 24 row window. And just as you can switch formats from the keyboard or during the execution of a program, you can also adjust the size of the print window. Adjusting the window means adjusting the position of the first and last column of text, and the position of the first and bottom row of text. You do this by POKEing the appropriate values into a family of four special RAM addresses.

**Setting the Starting Column of Text** Recall that Fig. 2-1 showed each line of text organized into 40 different column locations that were designated 0 through 39. Column address 0 represented a character location at the extreme left-hand edge of the screen, and column address 39 represented a character location at the extreme right-hand edge.

There is a particular memory location in Apple RAM that determines the column address for the first character to be printed on each line. That memory location, called WNDLFT, is at decimal address 32. Thus any reference to WNDLFT (address 32) has something to do with the left-hand starting position of a line of text.

Normally the value stored in WNDLFT is zero. That means each line of text will begin from column location 0—the extreme left-hand edge of the screen. But you can alter the value saved in WNDLFT at will.

Try this experiment:

1. Working in the Integer BASIC command mode, do a RESET followed by CTRL-C or CTRL-B. This action makes certain that the text window is set to its normal, maximum size.
2. Enter the following:

```
PRINT PEEK(32)
```

This PEEKs into WNDLFT and prints the value stored in it, which is 0.



3. Enter:

```
POKE 32,15
```

You will see the cursor and prompt symbols jump immediately to a position near the middle of a line. Do several RETURN keystrokes in succession to confirm that each line of text is beginning at column address 15.

4. Enter:

```
PRINT PEEK(32)
```

The response should be a 15 printed on the screen. This tells you that Step 3 was successful.

5. Enter:

```
POKE 32,0
```

This returns the value in WNDLFT to that required for the normal, 40-character-per-line format. A few keystrokes and RETURN operations will confirm that fact. Doing a PRINT PEEK(32) should turn up a value of 0.

You can POKE a lot of other positive integer values into WNDLFT, but only those in the range of 0 through 39 have meaning. When experimenting with the content of WNDLFT, you should not overflow a line of text when using a value other than 0. We can get away with it here, but allowing a line to overflow upsets the general scheme and, more importantly, forces data into RAM locations that aren't supposed to be affected by text operations.

The next section of this chapter describes how to avoid the problems inherent in overflowing a line when WNDLFT has some value other than 0 stored in it.

**Setting the Number of Characters Per Line** A special Apple RAM location called WNDWIDTH carries the number of characters that can be printed on each line. WNDWIDTH is at address 33, and it normally carries a value of 40. You can work with WNDWIDTH in much the same way as you did with WNDLFT.

1. Do a RESET followed by a CTRL-C or CTRL-B. This sets up the normal text window and BASIC's command mode.
2. Enter:

```
PRINT PEEK(32),PEEK(33)
```

This PEEKs into WNDLFT and WNDWDTH, respectively, and prints the values contained in them. Since the system is now set for the normal text window, the values should be 0 and 40. This means that the text begins at column address 0 on each line and that there are a maximum of 40 characters per line.

3. Enter:

```
POKE 33,10
```

This operation sets WNDWDTH to carry a value of 10. Type in a long string of arbitrary characters, and you will find that you can fit no more than 10 characters on each line.

4. Enter:

```
PRINT PEEK(32),PEEK(33)
```

The response should be a 0 in WNDLFT and a 10 in WNDWDTH. The text thus begins at column address 0 on each line with up to 10 characters per line.

5. Enter:

```
POKE 33,40
```

This should return the system to the normal 40-character-per-line text window. Confirm that by typing in long lines of text or by executing the command in Step 4.

WNDLFT and WNDWDTH are used in conjunction with one another to set the horizontal position and width of the text window. The possibility of causing serious problems by overflowing a line of text can be eliminated by a careful selection of values POKED into WNDLFT and WNDWDTH.

To avoid line overflow and possible destruction of data in other RAM locations, the sum of the values stored in WNDLFT and WNDWDTH must be less than or equal to 40.

Thus the following set of POKEs into WNDLFT and WNDWDTH make up a legitimate adjustment of the text window:

```
POKE 32,10:POKE 33,15
```

The first statement sets WNDLFT value to 10 so that text printing begins at the column-10 location on each line. The second statement POKEs 15 into

WNDWDTH, setting up the text for no more than 15 characters per line. The text window, in other words, occupies column addresses 10 through 24.

Quite often a programmer who is setting up a custom text window is more conscious of the column addresses at the beginning and end of each line (as opposed to the column address of the first character and the number of characters per line). A simple equation can keep things straight:

$$\text{WNDWDTH} = \text{WNDRT} - \text{WNDLFT} + 1$$

where,

WNDWDTH is the content of address WNDWDTH,

WNDLFT is the content of address WNDLFT,

WNDRT is the column address for the last character on each line.

Suppose that you want to configure a text window in such a way that the first character on each line is at column address 12 and the last character is at column address 36. Entering a POKE 32,12 sets WNDLFT to the appropriate value. Solving the equation provides the value to be POKED into WNDWDTH:

$$\text{WNDWDTH} = 36 - 12 + 1 = 25$$

Entering POKE 33,25 completes the task.

**Setting the Position of the Top Line** The normal text window allows up to 24 lines of text. As shown back in Fig. 2-1, those rows of text can be assigned addresses of 0 through 23, with row 0 representing the top row.

The row address of the topmost line of text is carried in Apple RAM by WNDTOP, or address 34. The system usually sets the content of WNDTOP to 0, indicating that the first line of text should appear at the very top of the screen.

Try this experiment:

1. Do a RESET followed by a CTRL-C or CTRL-B. This sets the text window for its normal 24-line format.
2. Enter:

**PRINT PEEK(34)**

The system should respond by printing a 0. The command PEEKs into WNDTOP and prints the content for you. In the normal text window, the content of WNDTOP is 0.

3. Enter:

POKE 34,10

This should set up the text window so that the first line is at row address 10. But it is unlikely that you will see the cursor symbol snap to that line position. Recall from discussions in the previous chapter that the system doesn't set up cursor row parameters until it is forced to do so. One way to force the issue in this case is to enter a CALL -936 to home the cursor and clear the screen.

4. Enter:

CALL -936

This homes the cursor, but now you will find that home is no longer the upper left-hand corner of the screen. Rather, it is located at the beginning of row address 10—the value for WNDTOP entered in Step 3. What's more, you might find that this operation clears only that part of the screen from row address 10 to the bottom. If you had some text printed in row addresses 0 through 9, it would not have been disturbed by the home-and-clear command.

5. Type in some keystrokes, including a lot of RETURNS. You will notice that all text operations, including the scrolling effect, take place from row address 10 through the bottom of the screen.
6. To set up WNDTOP, enter:

POKE 34,20  
CALL -936

Now the text window begins at row address 20.

7. Return to the normal text window by entering:

POKE 34,0

or by performing a RESET followed by CTRL-C.

8. Assure yourself that the content of WNDTOP is now 0 by entering:

PRINT PEEK(34)

9. Enter:

GR

This sets up the Apple low-resolution graphics mode.

10. Enter:

**PRINT PEEK(34)**

PEEKing into WNDTOP while in the mixed low-resolution graphics mode should show a value of 20. The Apple normally allows four lines of text at the bottom of the low-resolution screen, so it should come as no surprise that the Apple sets the content of WNDTOP to 20 in that operating mode.

11. Enter:

**TEXT**

This returns the system to the text mode.

12. Enter:

**PRINT PEEK(34)**

You should find that the system has returned the content of WNDTOP to 0.

There are 24 lines of text that can be labeled 0 through 23 for the purposes of setting the value in WNDTOP. You will discover after some experimenting, however, that WNDTOP works as described only for values 0 through 22. You can POKE a value of 23 into WNDTOP, but things get messed up if you do that, because the Apple expects the bottom line to be different than the top line.

Values stored in WNDTOP are limited to the range of positive integers from 0 through 22.
--

**Setting the Position of the Bottom Line** Apple RAM address 35, usually called WNDBTM, carries the position of the text window's bottom line. A value of 24 in WNDBTM allows the last line of text to appear at the very bottom of the screen.

The useful range of values that can be POKED into WNDBTM is between 1 and 24. Clearly the value of WNDBTM is not exactly represented by the usual row addressing scheme that runs between 0 and 23. Most Apple references cite the value of WNDBTM as being the desired row address of the bottom line plus 1. The notion is a valid one, of course, but it doesn't point to the rationale behind it. Consider this idea instead:

Number of lines=WNDBTM-WNDTOP

where,

WNCBDM is the content of WNCBDM,  
WNCBDM is the content of WNCBDM.

If you happen to POKE a value of 20 into WNCBDM and a 12 into WNCBDM, the equation says that you ought to end up with a text window that has just eight lines in it. In other words, the difference in line-numbering makes it easier to find the number lines in the text window.

Try it for yourself:

1. Enter:

```
POKE 34,12
POKE 35,20
CALL -936
```

This sets up the text window just described.

2. Type in some arbitrary characters, including some RETURN key-strokes. You will find that the printing and scrolling operations are confined to eight lines.
3. Enter:

```
PRINT PEEK(34),PEEK(35)
```

This PEEKs into WNCBDM and WNCBDM, respectively. The system should respond by printing 12 and 20.

4. Return to the full 24-line text window by entering:

```
TEXT
```

or

```
POKE 34,0
```

or

```
POKE 35,22
```

or by performing a RESET followed by CTRL-C. Any one of those three techniques will return WNCBDM and WNCBDM to their normal values of 0 and 22, respectively.

There are many programming situations, however, in which the programmer is more aware of the desired row addresses for the top and bot-

tom of the screen than the value to be stored in WNCBTM. A simple algebraic rearrangement of the equation puts matters into a more direct light:

$$WNCBTM = \text{Number of lines} + WNCBTM$$

Suppose, then, that you want to set up a 10-line text window beginning from row address 2. What value should be loaded into WNCBTM? From the equation just cited, you should POKE a value of 10+2, or 12 into WNCBTM.

Enter the following sequence to set up that particular text window:

```
POKE 34,2
POKE 35,12
CALL -936
```

Try it yourself.

**Programming the Text Window** Table 3-2 summarizes the RAM addresses, names and range of values for the four text window parameters. The following examples demonstrate how to manipulate all four of them to achieve a desired result.

*Example 1.* Prepare a BASIC program that sets up a text window having these specifications:

- a. Twelve characters wide, beginning at column address 0.
- b. Eight lines long, beginning at row address 0.

**Table 3-2. RAM Names for Setting Text Window Sizes**

RAM	Meaning	RAM Address	Range of Values	Normal Value
WNCBFT	Column address of the first character in each line	32	0-39	0
WNCBWDTH	Number of characters in each line	33	1-40	40
WNCBTOP	Row address of the top line	34	0-22	0
WNCBTM	Number of lines + WNCBTOP	35	1-24	24

The specifications, presented in that fashion, lead directly to the values to be POKEd into WNDLFT, WNDWIDTH, and WNDTOP:

```
POKE 32,0
POKE 33,12
POKE 34,0
```

An equation cited in the previous section leads to the value to be POKEd into WNDBTM:

$$\begin{aligned}\text{WNBDM} &= \text{Number of lines} + \text{WNBDM} \\ \text{WNBDM} &= 8 + 0 \\ \text{WNBDM} &= 8\end{aligned}$$

So POKE 35,8 is part of the program as well.

Here is a program that will do the job:

---

```
10 CALL -936
20 POKE 32,0: POKE 33,12
30 POKE 34,0: POKE 35,8
40 CALL -936
50 END
```

---

Enter and run the program. You will find that the text window is indeed confined to a 12-column, 8-line format in the upper left-hand corner of the screen.

The main purpose of the home-and-clear statement in line 10 is to clear the entire screen before confining operations to the custom text window. The second home-and-clear statement in line 40 gets the cursor to the newly established home location. If, for any reason, you do not want to clear the entire screen prior to setting up the smaller text window, simply delete program line 10. That will leave all of the text that is outside the smaller window intact.

*Example 2.* Suppose that while planning a relatively complex text-oriented program, you find that you need a small text window running between column addresses 26 and 34, and between row addresses 18 and 22. Prepare a BASIC routine for setting up that text window without disturbing the text that might appear elsewhere on the screen.

The given specifications lead directly to the values to be POKEd into WNDLFT and WNDTOP:

```
POKE 32,26
POKE 34,18
```



This equation provides the value to be loaded into WNDWDTH:

$$\begin{aligned}\text{WNDWDTH} &= \text{WNDRT} - \text{WNDLFT} + 1 \\ \text{WNDWDTH} &= 34 - 26 + 1 \\ \text{WNDWDTH} &= 9\end{aligned}$$

And the value to be POKed into WNDBTM is equal to the row address of the bottom line plus 1:

$$\begin{aligned}\text{WNCBTM} &= \text{row address} + 1 \\ \text{WNCBTM} &= 22 + 1 \\ \text{WNCBTM} &= 23\end{aligned}$$

So the desired BASIC routine should look like this:

---

```
10 POKE 32,26: POKE 33,9
20 POKE 34,18: POKE 35,23
30 CALL -936
```

---

*Example 3.* Write a BASIC routine that restores any custom text window to the normal, full-screen format.

Here is one approach:

---

```
10 POKE 32,0: POKE 33,40
20 POKE 34,0: POKE 35,24
```

---

The idea is to POKE the normal values into WNDLFT, WNDWDTH, WNDTOP, and WNCBTM, respectively.

But there is a far simpler way to accomplish the same thing:

## 10 TEXT

Remember that the TEXT statement automatically restores the normal text window values. Given the choice, why not opt for the simpler approach?

Knowing how to program custom text windows ought to lead one to begin thinking in terms of split-screen text activity—preparing programs that run two or more custom text windows. Listing 3-2 is an example of such an application.

Enter this program and run it. You will first see ENTER A 1-LINE MESSAGE and the flashing cursor symbol just below the middle of the screen—at row addresses 16 and 17 to be exact.

### Listing 3-2. Split-Screen Example.

---

```
10 DIM M$(32)
20 TOP1=0:BOT1=8
30 TOP2=16:BOT2=18
40 TOPW=34:BOTW=35
50 CV=37:VPOS=0
60 CALL -936
100 GOSUB 1000
110 PRINT "ENTER A 1-LINE MESSAGE"
120 INPUT M$
130 GOSUB 2000
140 PRINT M$
150 VPOS= PEEK (CV)
160 GOTO 100
1000 POKE TOPW,TOP2: POKE BOTW,BOT2
1010 CALL -936
1020 RETURN
2000 POKE TOPW,TOP1: POKE BOTW,BOT1
2010 POKE CV,VPOS: CALL -922: CALL -998
2020 RETURN
```

---

Respond by typing in a message string up to 32 characters long. End this phase of the program by striking the RETURN key.

Immediately after that, you will see your message printed at the top of the screen; then the ENTER message and flashing cursor will appear once again just below the middle of the screen.

The next string you enter will appear below the first, the next string after that will appear on the third line from the top, and so on. As you continue entering messages, they appear in sequence at the top of the screen, scrolling upward as you exceed eight message strings.

Just from observing the behavior of this program you should get the idea that it is using two different text windows: a two-line text window for entering the current message, and an eight-line text window at the top of the screen for accumulating the series of messages. The upper text window is allowed to scroll, but the lower one is not.

This is an example of a two-window format that splits the screen horizontally. Now let's look at the listing more closely.

Line 10 dimensions string variable M\$ to carry up to 32 characters.

Line 20 sets the WNDTOP and WNDBTM values for the upper text window. TOP1=0 sets the value in WNDTOP for the upper text window, and BOT1=8 sets the value in WNDBTM for that upper window.

Line 30 sets the WNDTOP and WNDBTM values for the lower text window. TOP2=16 sets the value to be POKEd into WNDTOP for the lower text window, and BOT2=18 sets the value in WNDBTM for the lower text window.

Line 40 points to the RAM addresses for WNDTOP and WNDBTM with variables TOPW and BOTW. (Integer BASIC does not accept WNDTOP and WNDBTM as variable names.)

Line 50 points to the address of RAM location CV, which carries the current row address of the cursor. It initializes variable VPOS, which will keep track of the cursor's vertical position in the upper text window while the program is running operations in the lower window.

Line 60 clears the screen and homes the cursor.

Line 100 goes to subroutine 1000 to set the text window parameters for working in the lower window.

Line 110 prints the prompt message.

Line 120 inputs the message to be printed in the upper window.

Line 130 goes to subroutine 2000 to readjust the window parameters for working in the upper text window.

Line 140 prints the message in the upper text window.

Line 150 saves the current cursor line number in VPOS.

Line 160 loops back to program line 100 to repeat the sequence.

Line 1000 is the beginning of a subroutine that sets up the lower text window. It begins by POKEing TOP2 into TOPW (16 into WNDTOP) and BOT2 into BOTW (18 into WNDBTM).

Line 1010 homes the cursor within the lower window, and clears that window.

Line 1020 returns to the mainline program.

Line 2000 marks the beginning of a subroutine that sets up operations for the upper text window. It begins by POKEing TOP1 into TOPW (0 into WNDTOP) and BOT1 into BOTW (8 into WNDBTM).

Line 2010 sets the cursor at the position it held after printing the previous message in the upper text window.

Line 2020 returns to the mainline program.

You can see that the program is divided into three main parts. Lines 10 through 60 initialize the program and define the variables. Lines 100 through 160 make up the mainline portion of the program. Finally, lines 1000 through 1020 and 2000 through 2020 are the window-setting subroutines.

You can change the size and relative positions of these two text windows by altering the values assigned to the windowing variables in lines 20 and 30. Try that now. You can also extend the scheme to include more than two such windows. Simply extend the assignments of variables, add more window-setting subroutines, and rewrite the main portion of the program to work in the new windows.

The next program, Listing 3-3, uses the above principles to split the screen into two vertical windows. Enter the program, do a RUN, and play

with it for a while. You should be able to analyze the listing on your own, using the analysis of the previous program as a general guide.

### **Listing 3-3. Second Split-Screen Example.**

---

```
10 DIM M$(32)
20 LFT1=0:NCH1=19
30 LFT2=20:NCH2=16
40 LWND=32:RWND=33
50 CV=37:VPOS=0
60 TEXT : CALL -936
100 GOSUB 1000
110 POKE 50,63: PRINT "ENTER A MESSAGE"
120 POKE 50,255
130 INPUT M$
140 GOSUB 2000
150 PRINT M$
160 VPOS= PEEK (CV)
170 GOTO 100
1000 POKE LWND,LFT2: POKE RWND,NCH2
1010 CALL -936
1020 RETURN
2000 POKE LWND,LFT1: POKE RWND,NCH1
2010 POKE CV,VPOS: CALL -922: CALL -998
2020 RETURN
```

---

When the need arises, you should be able to devise text-oriented programs that combine two or more text windows in both a horizontal and vertical fashion. It's all a matter of setting up the values for WNDLFT, WNDWDTH, WNDTOP, WNDBTM, and, on occasion, CV.

# Poke to Video Memory

## 4

The BASIC PRINT statement is the most commonly used tool for printing text on the Apple screen. But that isn't the only way to print characters. The POKE statement also does the job. POKE isn't the simplest way to print text, but it does offer some advantages under certain circumstances. For example, some characters cannot be printed on the screen except through POKE. Some of these characters do not even appear on the keyboard. POKE can also display all characters in flashing format. PRINT, you'll recall, cannot.

Finally, there is the matter of working with the secondary text page—an entire screen that is available for alternate blocks of text. The secondary screen is not readily accessible from Integer BASIC, but it is wide open to POKE-type printing operations.

**ORGANIZATION OF THE TEXT MEMORIES** There is a direct, one-for-one correspondence between each of the 960 character locations on the screen and each of 960 address locations in RAM. POKEing a character code into one of those RAM locations makes that character appear at a well-defined place on the screen. So, generally speaking, POKEing characters onto the screen is a matter of POKEing their character codes into the video text RAM area.

Table 4-1 is a memory map of the video text memory for the primary page. Generally speaking, it occupies most of the RAM addresses from 1024 to 2039.

Each line, taken alone, has a very logical and systematic allocation of RAM addresses. Line 0, for example, operates from addresses 1024 through 1063; address 1024 represents the first character in line 0, and address 1063 represents the last character in that same line. You would think that line 1—the next line down on the screen—would use RAM addresses 1064 through 1103. It doesn't. That range of 40 consecutive RAM addresses refers to line 8 on the screen. Strangely enough, the 80 consecutive RAM addresses between 1024 and 1103 are equally divided between two non-consecutive lines on the screen—between lines 0 and 8.

**Table 4-1. Primary Text Page**

Line Number	Video RAM Addresses
0	1024–1063
1	1152–1191
2	1280–1319
3	1408–1447
4	1536–1575
5	1664–1703
6	1792–1831
7	1920–1959
8	1064–1103
9	1192–1231
10	1320–1359
11	1448–1487
12	1576–1615
13	1704–1743
14	1832–1871
15	1960–1999
16	1104–1143
17	1232–1271
18	1360–1399
19	1488–1527
20	1616–1655
21	1744–1783
22	1872–1911
23	2000–2039

Where is the next sequence of 40 RAM locations used? Line 16! It uses the next series of 40 RAM addresses, 1104 through 1143.

So the sequences of RAM addresses begin at line 0, go to line 8, and then to line 16. Now which line does address 1144 refer to? You won't find that address on this memory map! RAM address 1144 isn't even part of the video text environment. In fact, the next video text address is 1152 at the beginning of line 1. This is because eight of the RAM addresses are used as I/O slots.

The next series of addresses is divided among text lines 1, 9, and 17. Then there is another gap of eight addresses, allocated for I/O slots. You will find the third series of addresses, 1280 through 1399, allocated for text lines 2, 10, and 18. It goes on like that through the entire text scheme.

It would seem to be enough to drive a sane programmer crazy. But take heart in the fact that there are some mechanisms for dealing with this

seemingly awkward arrangement of the video memory, which we will discuss later.

You will find the same sort of arrangement in the memory map of the secondary text page shown in Table 4-2. The only difference is that this map uses addresses 2048 through 3063. (Line-for-line, that means the secondary page addresses are equal to the primary page addresses plus 1024. This will be an important notion to remember later in this chapter.)

Try this short program to see how it is possible to POKE characters to the screen:

---

```
10 CALL -936
20 FOR N=0 TO 9
30 POKE 1024+N,112+N
40 NEXT N
50 END
```

---

After running that program, you will see that it is entirely possible to print flashing numerals. This cannot be done with a simple PRINT statement, even with the system set up for printing flashing characters.

The operation of this program is simple. Line 10 homes the cursor and clears the screen. There is a 0-to-9 FOR-NEXT loop between lines 20 and 40. Finally, line 30 POKES character codes from 112 to 121 into addresses 1024 through 1033.

The following program works just like the previous one, but POKES 120 successive character codes into 120 successive video addresses.

---

```
10 CALL -936
20 FOR N=0 TO 119
30 POKE 1024+N,N
40 NEXT N
50 END
```

---

This program prints three full lines of text on lines 0, 8, and 16. Now it is time to take a closer look at the character codes.

**VIDEO CHARACTER CODES** Every screen-printable character has three code numbers assigned to it. The three numbers represent the three text formats: inverse, flashing, and normal. Altogether, there are 256 different character codes ranging from 0 through 255.

**Inverse Character Codes** Table 4-3 shows the Apple characters and the corresponding codes for printing them in an inverse, black-on-

**Table 4-2. Secondary Text Page**

Line Number	Video RAM Addresses
0	2048–2087
1	2176–2215
2	2304–2343
3	2432–2471
4	2560–2599
5	2688–2727
6	2816–2855
7	2944–2983
8	2088–2127
9	2216–2255
10	2344–2383
11	2472–2511
12	2600–2639
13	2728–2767
14	2856–2895
15	2984–3023
16	2128–2167
17	2256–2295
18	2384–2423
19	2512–2551
20	2640–2679
21	2768–2807
22	2896–2935
23	3024–3063

white format. The codes range from 0 for an inverse @ symbol through 63 for an inverse question mark.

Here is a program that POKes those characters onto the screen for you:

---

```
10 CALL -936
20 FOR N=0 TO 63
30 POKE 1024+N,N
40 NEXT N
50 END
```

---

The program first homes the cursor and, what is more important, clears the screen. After that, it POKes the inverse-text character codes 0 through 63 into RAM addresses 1024 through 1087.



**Table 4-3. Inverse Screen Text Codes and Characters**

Code	Character	Code	Character	Code	Character	Code	Character
0	@	16	P	32		48	0
1	A	17	Q	33	!	49	1
2	B	18	R	34	"	50	2
3	C	19	S	35	#	51	3
4	D	20	T	36	\$	52	4
5	E	21	U	37	%	53	5
6	F	22	V	38	&	54	6
7	G	23	W	39	'	55	7
8	H	24	X	40	(	56	8
9	I	25	Y	41	)	57	9
10	J	26	Z	42	*	58	:
11	K	27	[	43	+	59	;
12	L	28	/	44	,	60	<
13	M	29	]	45	-	61	=
14	N	30	^	46	.	62	>
15	O	31	—	47	/	63	?

**Flashing Character Codes** A flashing character is one that alternates between inverse and normal display. The family of flashing characters can be displayed by POKEing their character codes into video text memory. (See the flashing character set and their respective codes in Table 4-4.)

The 64 flashing-character codes begin where the inverse-character codes end, running from 64 through 127. The following program lets you POKE these codes to the text screen:

---

```

10 CALL -936
20 FOR N=0 TO 63
30 POKE 1024+N,N+64
40 NEXT N
50 END

```

---

**The Normal Text Codes** The Apple character generator has two sets of normal, white-on-black, characters and character codes. For the sake of keeping things straight, we have labeled the two groups NORMAL-1 and NORMAL-2. (See Tables 4-5 and 4-6.)

The NORMAL-1 family of text characters uses codes 128 through 191, and the NORMAL-2 family uses codes 192 through 255. The reasons for the differences in code numbers for two otherwise identical characters

**Table 4-4. Flashing Screen Text Codes and Characters**

Code	Character	Code	Character	Code	Character	Code	Character
64	@	80	P	96		112	0
65	A	81	Q	97	!	113	1
66	B	82	R	98	"	114	2
67	C	83	S	99	#	115	3
68	D	84	T	100	\$	116	4
69	E	85	U	101	%	117	5
70	F	86	V	102	&	118	6
71	G	87	W	103	'	119	7
72	H	88	X	104	(	120	8
73	I	89	Y	105	)	121	9
74	J	90	Z	106	*	122	:
75	K	91	[	107	+	123	;
76	L	92	/	108	,	124	<
77	M	93	]	109	-	125	=
78	N	94	^	110	.	126	>
79	O	95	—	111	/	127	?

aren't important at this time. It is sufficient to say that POKEing 129 to the video memory and 193 to video memory both turn up a normal-format character A on the screen.

The following program POKes the NORMAL-1 family of characters to the screen:

---

```

10 CALL -936
20 FOR N=0 TO 63
30 POKE 1024+N,N+128
40 NEXT N
50 END

```

---

And this next one POKes the NORMAL-2 characters to the screen:

---

```

10 CALL -936
20 FOR N=0 TO 63
30 POKE 1024+N,N+192
40 NEXT N
50 END

```

---

**Table 4-5. NORMAL-1 Screen Text Codes and Characters**

Code	Character	Code	Character	Code	Character	Code	Character
128	@	144	P	160		176	0
129	A	145	Q	161	!	177	1
130	B	146	R	162	"	178	2
131	C	147	S	163	#	179	3
132	D	148	T	164	\$	180	4
133	E	149	U	165	%	181	5
134	F	150	V	166	&	182	6
135	G	151	W	167	'	183	7
136	H	152	X	168	(	184	8
137	I	153	Y	169	)	185	9
138	J	154	Z	170	*	186	:
139	K	155	[	171	+	187	;
140	L	156	/	172	,	188	<
141	M	157	]	173	-	189	=
142	N	158	^	174	.	190	>
143	O	159	—	175	/	191	?

**Table 4-6. NORMAL-2 Screen Text Codes and Characters**

Code	Character	Code	Character	Code	Character	Code	Character
192	@	208	P	224		240	0
193	A	209	Q	225	!	241	1
194	B	210	R	226	"	242	2
195	C	211	S	227	#	243	3
196	D	212	T	228	\$	244	4
197	E	213	U	229	%	245	5
198	F	214	V	230	&	246	6
199	G	215	W	231	'	247	7
200	H	216	X	232	(	248	8
201	I	217	Y	233	)	249	9
202	J	218	Z	234	*	250	:
203	K	219	[	235	+	251	;
204	L	220	/	236	,	252	<
205	M	221	]	237	-	253	=
206	N	222	^	238	.	254	>
207	O	223	—	239	/	255	?

You will see no difference between the two displays.

Of course if you want to see the entire Apple character set POKEd to the screen, you can run the following version of the previous programs. It POKEs character codes all the way through the set—from 0 through 255.

---

```
10 CALL -936
20 FOR N=0 TO 255
30 POKE 1024+N,N
40 NEXT N
50 END
```

---

Notice that characters running past the end of the first line on the screen do not continue at the beginning of the second line. This program POKEs into successively higher video RAM addresses, but the line format on the screen doesn't follow a line-by-line format. The next section of this chapter deals with that problem in some detail.

**GETTING SOME HELP FROM THE MONITOR** According to Table 4-1, the last character in the first line of text is always loaded into RAM address 1063. Try this:

**POKE 1063,32**

That should POKE a white rectangle, or inverse space character, into the last character location on the top line of the screen.

Now try this:

**POKE 1064,32**

Does this command POKE the white rectangle to the beginning of the second line of text? After all, you've used the next-higher RAM address location. But as you can see, the white rectangle doesn't appear on the second line. Instead, it appears at the beginning of a line somewhat lower on the screen.

Referring back to Table 4-1, you will see that the beginning of the second line on the screen is represented by video RAM address 1152. Try this:

**POKE 1152,32**

That does the job; but you have to look up the starting address of each line to POKE texts. Or so it would seem.

The question of where to POKE a character code becomes especially

---

important when attempting to POKE in a long string of characters that are to occupy more than one line in succession. The most obvious trouble is that you have to adjust the POKE address sequence to begin the next line. A less obvious, but no less important, problem is that there are groups of RAM addresses in the video text memory that have nothing at all to do with video text. If you are careless about setting up the POKE addresses, you will end up POKEing character data into those places; and that risks upsetting the parameters for some important I/O functions.

As an extreme case, suppose that you want to fill the entire screen with 960 white rectangles. And what's more, you want to POKE them onto the screen in a left-to-right, top-to-bottom sequence.

Looking over the video memory map in Table 4-1, you find that the addresses cover the range of 1024 through 2039. Anyone unfamiliar with the unusual arrangement of addresses for the video memory might suppose the program would look like this:

---

```
10 CALL -936
20 FOR N=1024 TO 2039
30 POKE N,32
40 FOR T=0 TO 10: NEXT T
50 NEXT N
60 GOTO 60
```

---

That program fills the screen with white rectangles, but not in a sequential top-to-bottom fashion, even though it POKES to RAM addresses in a strict sequential fashion. And what's more, there are some I/O ports being given character codes that they should not be receiving.

It's a problem, but there is a solution. The software designers for the Apple company had to deal with the same problem, and they responded to it by including some useful subroutines in the monitor ROM. Those subroutines, which we'll discuss in greater detail later, take care of the sticky task of calculating the RAM addresses for sequential plotting of characters on the screen. We can use those subroutines to make the POKE-to-video task a lot simpler for us.

The following program uses those subroutines to fill the screen with white rectangles in a left-to-right, top-to-bottom fashion. It doesn't POKE data into the I/O buffer spaces, either.

Notice that there are no references to absolute video memory address locations. Indeed, this program shows that it is possible to do POKE text operations without worrying about the RAM address of the next line of text. (By the way, the address of the next line is considered to be the address of the first character location in that line. This address is sometimes called the base address of the line.)

---

```
10 CALL -936
20 FOR N=0 TO 959
30 PT=256* PEEK (41)+ PEEK (40)+ PEEK (36)
40 POKE PT,32
50 FOR T=0 TO 10: NEXT T
60 CALL -1036
70 NEXT N
80 GOTO 80
```

---

**The Role of BASL and BASH** The Apple system uses two RAM locations to keep track of the base address of the current line of text. The RAM locations assigned to that task are called BASL and BASH, and they are at RAM addresses 40 and 41, respectively. Try this:

```
PRINT PEEK(40),PEEK(41)
```

and you will see two numbers printed on the screen. They represent the actual video memory address of the first column in the current row of text.

Why two numbers? Because the 1-byte memory capacity of each address location in the Apple can hold values no larger than 255. The video memory addresses are much larger than that—1024 for the first character in the first row, for instance. Thus it takes two bytes of memory to represent those addresses.

BASL carries the lower-order byte, and BASH carries the higher-order byte of the video line address. You can come up with a more meaningful address number, then, by doing something such as this:

```
PRINT 256*PEEK(41)+PEEK(40)
```

The result is a number that designates the video memory address of the first character in the *current* line of text. It will be one of the start-of-line addresses shown in Table 4-1. (See Appendix A if you are not sure about the technique for converting a decimal number spread out over 2 bytes into a single decimal number.)

**The Role of CH** You have just seen that BASL and BASH provide the absolute RAM address of the *first* character location in the current line of text. That is a useful figure, of course, but suppose you wanted to find the RAM address of the *current* character location. For that you would have to add the column number of the current character location to the base address of the line.

BASL and BASH carry the video memory address of the first character in the current line of text.

BASL, at address 40, carries the least-significant byte.

BASH, at address 41, carries the most-significant byte.

PRINT 256\*PEEK(41)+PEEK(40) returns the actual, full-decimal value of the video memory address of the first character in the current line of text.

Recall from Chapter 2 that a memory location called CH, at address 36, carries the current column address of the cursor. By definition, the cursor has the same column address as the current character location. Therefore, finding the full POKE address of a character location is a matter of adding the content of CH to BASL and BASH. That is why line 30 in the last full program shown here reads like this:

**PT=256\*PEEK(41)+PEEK(40)+PEEK(36)**

This statement PEEKs into BASH and multiplies BASH's content by 256, PEEKs into BASL and CH, and sums everything together. The overall result is assigned to variable PT, but any variable could have been used.

As that white-line drawing program runs, variable PT takes on all of the values for addresses in the primary page video RAM. If you were to monitor the values of PT while running the program, you would find that they follow the sequence shown in Table 4-1. A subroutine in the monitor is taking care of setting up the RAM starting addresses for each new line of text.

**The Roles of ADVANCE and BASCALC** Recall that Chapter 2 described ADVANCE, a monitor subroutine that advances the cursor one column to the right. If the cursor is at the end of a line, ADVANCE sends the cursor to the beginning of the next line on the screen. Recall further that you can call the ADVANCE subroutine from BASIC with the statement CALL -1036.

You don't have to be working with the cursor to take advantage of ADVANCE. The ADVANCE subroutine simply increments the content of CH until the end of a line is reached. Once the content of CH indicates the end of a line, ADVANCE calls BASCALC, another subroutine that calcu-

lates the base address of the next line and places the address into BASL and BASH. ADVANCE then resets CH back to 0.

The program that filled the screen with white rectangles from left to right and from top to bottom used the ADVANCE subroutine to calculate the proper video memory addresses. Unfortunately, we cannot write a BASIC program that demonstrates BASCALC, because BASCALC cannot be called directly from BASIC. We will be able to demonstrate BASCALC later when we learn about assembly-language programming.

**The Importance of All of This** Why are we worrying about the complications of POKE addressing for the video text environment? Why not simply stay with the simpler PRINT and cursor-related operations? There are some good answers.

For one, you have already seen that you cannot PRINT flashing numerals and punctuation marks. POKEing to video memory offers that opportunity.

Second, you will soon find that POKE-oriented text operations offer the only decent way to work with the secondary page of video text from BASIC.

Finally, these text-POKEing techniques are practically identical to those machine-language programs featured later in this book.

Indeed, there is sufficient reason for this current series of discussions.

**BUILDING AND USING MESSAGE BLOCKS** This section shows you how to use what you've already learned to place blocks of characters on the screen.

The simplest way to get a message string printed to the screen is by means of a BASIC statement such as:

```
PRINT "HELLO"
```

But suppose that you want to POKE that same message to the screen. The general idea is to assign each of the five characters in HELLO to a different variable, and then POKE those variables to a selected portion of video RAM. Try this program:

First notice the sequence of character codes assigned to array variables C(0) through C(4) in program lines 130 and 140. If you compare them with the NORMAL-2 codes in Table 4-6, you will find that they make up the message string HELLO. The character code for the letter H is assigned to C(0), the code for E is assigned to C(1), and so on. Such a group of code assignments makes up a *message block*.

Program lines 150 through 170 run through the block of character



---

```
100 DIM C(64)
110 CALL -936
120 MP=1024
130 C(0)=200:C(1)=197:C(2)=204:C(3)=204
140 C(4)=207
150 FOR N=0 TO 4
160 POKE MP+N,C(N)
170 NEXT N
180 END
```

---

codes and POKE them in sequence to the video RAM and, hence, the screen. Variable MP sets the starting address of the message. The rest of the message is displayed by the summation operation in line 160, which increments the POKE addresses to place the message in a left-to-right sequence.

Altering the value assigned to MP changes the starting address for the message. Tinker with that variable, using Table 4-1 as a guide. Of course you should make sure that your starting addresses do not allow the message to overflow at the end of a line or POKE data outside the video RAM area.

Make up a short message of your own, look up the character codes for each character, and assign them to the C array. Add some life to the message by using some flashing and inverse characters. Just remember that the DIM statement in line 100 limits you to 65 characters.

**Using Multiple Message Blocks** Most practical message-POKEing situations call for working with more than one message block. The program in Listing 4-1, for example, includes three message blocks.

The first message block is carried by array variables C(0) through C(4) in program lines 1110 and 1120. That one spells out HELLO with NORMAL-2 character codes.

The second block, assigned to array variables C(0) through C(6) in lines 1210 and 1220, spells out BLOCK 2 with a flashing numeral 2. (That cannot be done with a simple PRINT statement.)

The third block, 3RD MESSAGE, is assigned to array variables C(0) through C(10) in lines 1310 through 1330.

Each of those message blocks is imbedded in a short subroutine. Each subroutine begins with assignment of a value to variable NC and ends with a RETURN. The message blocks, in other words, are written as subroutines that:

1. Assign a value to NC that represents the number of consecutive characters in the block.

#### Listing 4-1. Multiple Message Blocks.

---

```
100 DIM C(32)
110 CALL -936
120 MP=1024
130 GOSUB 1100: GOSUB 2000
140 MP=1080
150 GOSUB 1200: GOSUB 2000
160 MP=1104
170 GOSUB 1300: GOSUB 2000
180 END
1100 NC=4
1110 C(0)=200:C(1)=197:C(2)=204:C(3)=204
1120 C(4)=207
1190 RETURN
1200 NC=6
1210 C(0)=194:C(1)=204:C(2)=207:C(3)=195
1220 C(4)=203:C(5)=224:C(6)=114
1290 RETURN
1300 NC=10
1310 C(0)=243:C(1)=210:C(2)=196:C(3)=224
1320 C(4)=205:C(5)=197:C(6)=211:C(7)=211
1330 C(8)=193:C(9)=199:C(10)=197
1390 RETURN
2000 FOR N=0 TO NC
2010 POKE MP+N,C(N)
2020 NEXT N
2030 RETURN
```

---

2. Assign character codes to array variable C.
3. RETURN to the calling main routine.

The subroutine for the first block occupies lines 1100 through 1190; the subroutine for the second block occupies lines 1200 and 1290; and the subroutine for the third block occupies lines 1300 through 1390. The three subroutines are called from lines 130, 150, and 170, respectively. But it is not enough to simply assign the character codes, as these subroutines do—the codes have to be POKEd to the screen, too. That is the purpose of the subroutine beginning at line 2000.

The subroutine occupying program lines 2000 through 2030 prints the current message block to the video RAM and screen. It POKEs N+1 characters in sequence, beginning from address MP. Thus, line 130 sets up the number of characters to be printed, assigns the character codes to the array variables, and POKEs them to the screen. The MP=1024 assignment in line 120 sets up the program for starting the first message at video address 1024.

In a similar way, lines 140 and 150 set up and POKE the second message block to the screen; and lines 160 and 170 set up and POKE the third message to the screen.

The program is thus divided into three main sections:

1. A main routine (lines 100 through 180) that establishes the sequence of events and calls the appropriate subroutines.
2. The blocks of character codes for each message.
3. A POKEing subroutine that applies to all of the message-printing operations (lines 2000 through 2030).

This particular programming format makes it relatively easy to expand the number of message blocks and alter their sequence and positions for POKEing them to the screen. Add a few message-block subroutines of your own, and call and print them from an extended version of the main section of the program.

Listing 4-2 demonstrates how easy it is to change the entire operation of a block-printing program by altering only the main portion of it. This version lets you select the message block to be POKEd to the screen (program lines 20 to 50), and then prints it by selecting the appropriate setup routine in lines 110, 120, or 130.

Notice that the program combines ordinary PRINT operations with POKE-oriented printing operations.

What is the purpose of line 80? (Hint: Delete line 80 and see what happens when you run the program.)

**Altering the Character Format** Under certain circumstances, you might want to change the print format during the execution of a program. In the previous examples, for instance, you might have wanted to print the 3RD MESSAGE string as normal characters under some circumstances, but as inverse or flashing characters under other circumstances. To have done so would have been a simple matter of changing the POKE subroutine at line 2000. There would have been no need to reassign codes to the message blocks themselves.

The program in Listing 4-3 is an example of changing formats. It begins by POKEing the message, SYSTEM START, to the screen in the normal, white-on-black format. There is a short delay, after which the same message appears with START written in the inverse, black-on-white format.

Then the display changes to SYSTEM START—all in the normal format. After another delay, the START portion of the message changes to the inverse format, and later to the flashing format.

The program ends by displaying SYSTEM GO in the flashing format.

The program has just four message blocks: SYSTEM, START, READY, and GO. All are entered as NORMAL-2 characters.

## Listing 4-2. Choosing Message Blocks.

---

```
10 DIM C(32): CALL -936
20 PRINT "WHICH MESSAGE (1,2 OR 3)";
30 INPUT M
40 IF M=1 OR M=2 OR M=3 THEN 60
50 GOTO 20
60 CALL -936
70 GOSUB 100+10*M
80 TAB 1: VTAB 1: CALL -868
90 GOTO 20
110 MP=1080: GOSUB 1100: GOSUB 2000: RETURN
120 MP=1080: GOSUB 1200: GOSUB 2000: RETURN
130 MP=1080: GOSUB 1300: GOSUB 2000: RETURN
1100 NC=4
1110 C(0)=200:C(1)=197:C(2)=204:C(3)=204
1120 C(4)=207
1190 RETURN
1200 NC=6
1210 C(0)=194:C(1)=204:C(2)=207:C(3)=195
1220 C(4)=203:C(5)=224:C(6)=114
1290 RETURN
1300 NC=10
1310 C(0)=243:C(1)=210:C(2)=196:C(3)=224
1320 C(4)=205:C(5)=197:C(6)=211:C(7)=211
1330 C(8)=193:C(9)=199:C(10)=197
1390 RETURN
2000 FOR N=0 TO NC
2010 POKE MP+N,C(N)
2020 NEXT N
2030 RETURN
```

---

But the program also has four different POKE subroutines: normal character, inverse character, flashing character, and erase character.

It would be a good idea to enter and run the program before reading the following analysis of the subroutines.

Lines 1110 through 1490 are the message block subroutines. Specifically, lines 1110 through 1190 contain SYSTEM followed by a space; lines 1200 through 1290 contain START; lines 1300 through 1390 contain READY; and lines 1400 through 1490 contain GO. A GOSUB to one of these subroutines sets up the system for POKEing a message block.

Lines 2000 through 2330 are the POKE-to-screen subroutines. Specifically, lines 2000 through 2030 POKE NORMAL-2 characters; lines 2100 through 2130 POKE inverse characters; lines 2200 through 2230 POKE flashing characters; and lines 2300 through 2330 POKE blanks, or spaces. A GOSUB to one of these subroutines POKES the current message block to the screen.

The scheme assumes that the message blocks have NORMAL-2 char-

---

### Listing 4-3. Altering Character Format.

---

```
100 DIM C(16): CALL -936
110 MP=1064: GOSUB 1100: GOSUB 2000
120 MP=1071: GOSUB 1200: GOSUB 2000
130 GOSUB 900: GOSUB 2100: GOSUB 900: GOSUB 2300
140 MP=1071: GOSUB 1300: GOSUB 2000
150 GOSUB 900: GOSUB 2100: GOSUB 900
160 GOSUB 2200: GOSUB 900: GOSUB 2300
170 MP=1064: GOSUB 1100: GOSUB 2200
180 MP=1071: GOSUB 1400: GOSUB 2200
190 FOR N=1 TO 4: GOSUB 900: NEXT N
200 END
900 FOR T=0 TO 1500: NEXT T: RETURN
1100 NC=6
1110 C(0)=211:C(1)=217:C(2)=211:C(3)=212
1120 C(4)=197:C(5)=205:C(6)=224
1190 RETURN
1200 NC=4
1210 C(0)=211:C(1)=212:C(2)=193:C(3)=210
1220 C(4)=212
1290 RETURN
1300 NC=4
1310 C(0)=210:C(1)=197:C(2)=193:C(3)=196
1320 C(4)=217
1390 RETURN
1400 NC=1
1410 C(0)=199:C(1)=207
1490 RETURN
2000 FOR N=0 TO NC
2010 POKE MP+N,C(N)
2020 NEXT N
2030 RETURN
2100 FOR N=0 TO NC
2110 POKE MP+N,C(N)-192
2120 NEXT N
2130 RETURN
2200 FOR N=0 TO NC
2210 POKE MP+N,C(N)-128
2220 NEXT N
2230 RETURN
2300 FOR N=0 TO NC
2310 POKE MP+N,224
2320 NEXT N
2330 RETURN
```

---

acter codes. The POKE subroutine beginning at line 2000 works directly with NORMAL-2 codes. The inverse-character POKE subroutine at line 2100, however, subtracts 192 from the NORMAL-2 codes, thus generating character codes in the inverse format range. Likewise, the subroutine beginning at line 2200 subtracts 128 from the NORMAL-2 character codes, thus POKEing flashing characters to the screen.

---

The POKE-to-screen subroutine that begins at line 2300 ignores the characters in the message block. It simply POKEs spaces to the screen, thereby erasing the current message.

With those four message blocks and four POKE modes, you can structure a wide range of operations from the main program. We'll now make a detailed, line-by-line analysis of the main program:

Line 110 sets the starting video RAM address (MP), calls the SYSTEM message at line 1100, and then calls the POKE subroutine at line 2000 to print SYSTEM in normal format.

Line 120 sets the starting address for the second part of the message, calls the START message, and then calls the POKE subroutine at 2000 to print START in the normal format.

Line 130 calls the time delay subroutine at line 900, calls the subroutine at 2100 to print START in inverse format, calls the delay again, and then calls the erase subroutine.

Line 140 sets the starting address for the next message segment, calls the READY message, and then calls the subroutine at 2000 to POKE the message in the normal format.

Line 150 calls the time delay subroutine, the inverse character subroutine, and then calls the time delay subroutine again.

Line 160 calls the flashing character subroutine, the time delay subroutine, and then calls the erase subroutine.

Line 170 sets the starting address of a new message, calls the SYSTEM message again, and then calls the flashing character subroutine.

Line 180 sets the starting address of the second part of the message, calls the GO message, and then calls the flashing character subroutine.

Lines 190 and 200 calls the time delay subroutine four times in succession and then ends the program.

Test your understanding of this scheme by devising and running a main program of your own. By main program, of course, I mean the controlling portion of the program (lines 100 through 900 in this particular case).

**WORKING WITH THE SECONDARY TEXT PAGE** Table 4-2 is the memory map for a full second page of screen text. Usually called the *secondary page*, it follows the same general map as the primary page. The only difference is in the RAM addresses. Secondary-page RAM addresses begin where the primary-page addresses leave off.

The designers of the Apple system did not devise the Integer BASIC software with secondary-page applications in mind. Unless you take steps to remedy the situation, you will find that Integer BASIC uses the second-

ary page for storing values of variables. Integer BASIC, in other words, often stores valuable information in the secondary page, thus wrecking any attempt to control the characters POKEd into it. Also, if you try POKEing text characters into the secondary page, it is quite likely that you will disturb the operation of Integer BASIC itself.

But there is a simple remedy available: Do a LOMEM command from the keyboard to exclude BASIC from the secondary-page RAM. Set LOMEM to an address at the very top of the secondary page, and you will still have a lot of RAM available for BASIC. Doing a single LOMEM:3071 before starting the following series of demonstrations will do the job. All discussions in this section assume that you have done a LOMEM:3071 from the keyboard.

## Switching Between the Primary and Secondary Pages

While the Apple features two text memories, only one can be displayed on the screen at any given moment. Usually, that is the primary page. Here's how you can change that:

POKE -16299,0 displays the secondary page of text.  
POKE -16300,0 displays the primary page of text.

The primary page, the one mapped for you in Table 4-1, is the one the Apple system normally displays on the crt. You must do a POKE -16299,0 from the keyboard or in a BASIC program to get a look at the secondary page.

Try this series of experiments:

1. From Integer BASIC's command mode, type and enter:

```
LOMEM:3071  
POKE -16299,0
```

You will most likely see a lot of garbage on the screen. All of those characters represent the content of the secondary text page (which you probably haven't used since turning on the system today).

2. Type and enter:

```
POKE -16300,0
```

Surprise! You cannot see those characters as you type them. You are viewing the secondary page of text, and all interaction with BASIC

takes place on the primary page. In a manner of speaking, you have to “type in the dark” when viewing the secondary page.

But if you type and enter the command properly, you will find the system running with the primary page displayed once again. And there is your command written out for you.

Try this little demonstration program:

---

```
10 FOR T=0 TO 500: NEXT T
20 POKE -16299,0
30 FOR T=0 TO 500: NEXT T
40 POKE -16300,0
50 GOTO 10
```

---

The program switches the crt display between the primary and secondary pages of text, doing a short time delay between each display.

Two text memories and one display. That's the simple essence of the Apple text scheme. That BASIC uses portions of the secondary-page memory is easily fixed by LOMEMing BASIC out of that area. But there is another problem that isn't so easy to fix: all cursor operations refer only to the primary-page memory. And that means it is necessary to work with the secondary page by means of POKE-text techniques.

**Clearing the Secondary Page** The first step in most kinds of text-printing operations is to clear the screen. That is an easy task when working with the primary page. Simply do a CALL -936. That operation both homes the cursor and clears the primary-page RAM. There has to be some procedure for clearing the secondary page as well.

First, look at this procedure for clearing the primary page of text:

---

```
10 TAB 1: VTAB 1
20 FOR N=0 TO 959
30 POKE 256* PEEK (41)+ PEEK (40)+ PEEK (36),224
40 CALL -1036
50 NEXT N
60 END
```

---

Line 10 homes the cursor, and the remainder of the program POKes character 224 (a normal space) into all 960 character-location addresses in primary-page RAM. (See “Getting Some Help From the Monitor” on page 70 for a detailed explanation of program lines 30 and 40.)

That program, in its cumbersome fashion, clears the primary page of text. Look at the secondary page, however, and you will find that it is not affected. It is probably still filled with garbage.



Return to the primary page and modify line 30 to read like this:

```
30 POKE 256*PEEK(41)+PEEK(40)+PEEK(36)+1024,224
```

Run the program while viewing the primary page, and you will see nothing happening. There will be a delay before the program ends.

Now switch the display to the secondary page, and lo! it is cleared. If it isn't cleared, you have most likely forgotten to begin this series of demonstrations with a LOMEM:3071.

How does that routine work? Why does it affect only the secondary text? The change in line 30 of the original version of the clearing program adds 1024 to the current cursor location. The secondary page is formatted in exactly the same way as the primary page—the addresses are simply 1024 locations higher in memory. So the routine just cited clears either the secondary or primary text memory, depending on whether you add 1024 to the current cursor position or not.

Can you devise a way to modify the routine so that it clears both the primary and the secondary pages?

**POKEing Characters to the Secondary Page** Everything we've said about POKEing characters to the primary-page RAM area applies equally well to secondary-page operations. Just step up the text addresses by 1024. We'll repeat that:

Adding a value of 1024 to the address of any text operation that would normally affect the primary page will, instead, affect the secondary page.

Listing 4-4 is a secondary-page version of Listing 4-3. Compare Listings 4-3 and 4-4, and you will find that the latter includes a screen-clearing subroutine for the secondary page (program lines 1000 through 1050). Also, you will find the starting addresses for the message blocks (variable MP) are increased by 1024. The entire text, in other words, functions in the secondary page; and line 110 calls up that page so that you can view it.

Set up some message blocks, a POKE-to-screen routine, and tell the program to write your messages to the secondary screen. Just remember to:

1. Do a LOMEM:3071 if you haven't done so already.
2. Include a POKE -16299,0 to view the secondary page.
3. Include a screen-clearing routine for the secondary page.
4. POKE the character codes into the secondary page (Table 4-2).
5. Include a POKE -16300,0 wherever you want to return to the primary page.

#### Listing 4-4. Using the Secondary Page.

---

```
100 DIM C(16)
110 GOSUB 1000: POKE -16299,0
120 MP=2088: GOSUB 1100: GOSUB 2000
130 MP=2095: GOSUB 1200: GOSUB 2000
140 GOSUB 900: GOSUB 2100: GOSUB 900: GOSUB 2300
150 MP=2095: GOSUB 1300: GOSUB 2000
160 GOSUB 900: GOSUB 2100: GOSUB 900
170 GOSUB 2200: GOSUB 900: GOSUB 2300
180 MP=2088: GOSUB 1100: GOSUB 2200
190 MP=2095: GOSUB 1400: GOSUB 2200
200 FOR N=1 TO 4: GOSUB 900: NEXT N
210 POKE -16300,0: END
900 FOR T=0 TO 1500: NEXT T: RETURN
1000 TAB 1: VTAB 1
1010 FOR N=0 TO 959
1020 POKE 256* PEEK (41)+ PEEK (40)+ PEEK (36)+1024,224
1030 CALL -1036
1040 NEXT N
1050 RETURN
1100 NC=6
1110 C(0)=211:C(1)=217:C(2)=211:C(3)=212
1120 C(4)=197:C(5)=205:C(6)=224
1190 RETURN
1200 NC=4
1210 C(0)=211:C(1)=212:C(2)=193:C(3)=210
1220 C(4)=212
1290 RETURN
1300 NC=4
1310 C(0)=210:C(1)=197:C(2)=193:C(3)=196
1320 C(4)=217
1390 RETURN
1400 NC=1
1410 C(0)=199:C(1)=207
1490 RETURN
2000 FOR N=0 TO NC
2010 POKE MP+N,C(N)
2020 NEXT N
2030 RETURN
2100 FOR N=0 TO NC
2110 POKE MP+N,C(N)-192
2120 NEXT N
2130 RETURN
2200 FOR N=0 TO NC
2210 POKE MP+N,C(N)-128
2220 NEXT N
2230 RETURN
2300 FOR N=0 TO NC
2310 POKE MP+N,224
2320 NEXT N
2330 RETURN
```

# The Keyboard Environment

Just as the crt is the Apple's most-used output device, the keyboard is its most-used input device. The keyboard input environment isn't as sophisticated nor as versatile as the crt output environment, but it is no less useful in the hands of a programmer who really understands it.

5

In the immediate command mode of operation, the keyboard is normally linked directly to the video text display system. Just about every sort of keystroke produces some sort of response on the primary text screen, printing characters or doing one of several cursor-related functions, such as linefeeds, backspacings, and the like.

That normal link between the keyboard and video system is through the Apple port-0 Input/Output (I/O) slot. Turning on the computer or doing a RESET automatically activates that port-0 link.

If you have a printer attached to the system, however, you have the option of linking the keyboard to that printer, usually through the Port-1 I/O slot. Doing a PR#1 command then causes the keyboard operations to affect the printer instead of the video text system. Doing a PR#0 command returns the keyboard to the video text system again.

Of course the keyboard can be used as an input device during the execution of a program as well. The BASIC INPUT statement, for instance, halts the flow of a program, giving the user an opportunity to type in values for numeric or string variables. Striking the RETURN key resumes the execution of the program.

It is also possible to PEEK into the keyboard system. Unlike the INPUT statement, a properly designed PEEK-to-keyboard statement does not necessarily interrupt the flow of a program.

**SUPPLYING INFORMATION WITH INPUT** INPUT statements can be inserted into BASIC programs to give the user a chance to assign values to numeric or string variables. Upon encountering an INPUT statement, the system halts the normal step-by-step execution of the program

until the user strikes the RETURN key—usually after entering the appropriate information.

The general syntax of an INPUT statement is:

INPUT "*message*",*variable*

where *message* is an optional prompting message, and *variable* is a numeric or string variable that is to take on the value typed in by the user.

Here is a common INPUT situation:

INPUT "WHAT IS YOUR NAME?",N\$

On encountering that statement, BASIC responds by printing:

WHAT IS YOUR NAME?

It then shows the blinking cursor to signal the user it is time to type in a response. When the user enters a string response and strikes the RETURN key, the string is assigned to string variable N\$. N\$ then holds that string until some later operation calls for a change.

The type of variable used must match the type of information to be entered from the keyboard under the INPUT operation. In the previous example, the information to be entered from the keyboard was a string value—presumably your name. The variable that held that value, therefore, had to have been a string variable.

Here is an INPUT statement that expects a numeric input:

INPUT "WHAT NUMBER (0-9)",N

Encountering that statement, BASIC prints:

WHAT NUMBER (0-9)?

followed by the blinking cursor symbol. The user is then expected to type in a number and strike the RETURN key. From that point on, the number is assigned to variable N.

Whether or not an INPUT statement calls for a numeric or a string response, BASIC always inserts the flashing cursor symbol. But only the numeric INPUT statement has a question mark automatically inserted after the message. The string INPUT statement does not.

Suppose you want to prompt the user with a question in a string INPUT statement. Simply include the question mark in the message portion of the INPUT statement:

INPUT "WHAT IS YOUR NAME? ",N\$

The system responds by printing:

**WHAT IS YOUR NAME?**

followed by a space and the blinking cursor symbol.

Indeed, you have a lot of flexibility in phrasing prompting messages for instead of a question, simply omit the question mark from the message:

**INPUT "ENTER YOUR NAME ",N\$**

the system responds by printing:

**ENTER YOUR NAME**

followed by a space and the blinking cursor symbol.

Indeed, you have a lot of flexibility in phrasing prompting messages for string INPUT statements. There are some minor difficulties, though, in phrasing the prompting message for a numeric INPUT statement. The difficulties are caused by the automatic insertion of the question mark.

Suppose you want the user to type in a numeric value between 0 and 9. If you frame the prompting message without a question mark, as in:

**INPUT "ENTER A NUMBER BETWEEN 0 AND 9",N**

the system prints:

**ENTER A NUMBER BETWEEN 0 and 9?**

followed immediately by the cursor symbol. The automatically inserted question mark can be quite misleading. The uninformed user might interpret the prompting message as asking whether or not a number between 0 and 9 should be entered.

You must be careful about phrasing the prompting message. Here is a less confusing version:

**INPUT "WHAT NUMBER (0-9)",N**

In that case, the system prints:

**WHAT NUMBER (0-9)?**

followed by the cursor symbol.

The following is another way of doing the same thing:

```
100 PRINT "ENTER AN INTEGER VALUE BETWEEN 0 and 9"  
110 INPUT "WHAT NUMBER",N
```

The user then sees this on the screen:

```
ENTER AN INTEGER VALUE BETWEEN 0 and 9  
WHAT NUMBER?
```

That leaves little room for doubt about what the user is supposed to do.

An earlier comment in this chapter expressed the notion that it is possible to omit the message portion of an input statement, thus reducing it to either INPUT N\$ or INPUT N.

The first of those two INPUT statements is expecting a string input, and the second is expecting a numeric value. There is no message assigned to either of them, so the system responds to the first by showing only the blinking cursor, and to the second by showing a question mark followed by the blinking cursor.

The advantage of using these abbreviated forms is that they allow you to separate the prompting message from the blinking cursor, or question mark and cursor. Consider the following set of statements:

```
100 PRINT "WHAT IS YOUR NAME?"  
110 INPUT N$
```

In this case, the prompting message, WHAT IS YOUR NAME?, appears on one line, and the cursor symbol appears at the beginning of the next line on the screen.

Or try this:

---

```
10 DIM N$(15)  
20 CALL -936  
30 PRINT "ENTER YOUR NAME"  
40 TAB 20: VTAB 22  
50 INPUT N$  
60 TAB 1: VTAB 1  
70 CALL -868  
80 GOTO 80
```

---

Program line 20 both homes the cursor and clears the screen, and then line 30 prints the prompting message, ENTER YOUR NAME, in the upper left-hand corner. Line 40, however, sends the cursor down to the middle of

row address 21, so the INPUT statement is executed at that point. In other words, the blinking cursor and the user's keyboard entry appear there. The statements in line 60 return the cursor to home, and line 70 clears to the end of that line, erasing the prompting message. The user's key entry remains undisturbed near the bottom of the screen, however.

The general idea is to separate the prompting message and the user's keyboard response to the subsequent INPUT operation by a lot of text area—area that might include a lot of other text you might not want to disturb.

The troublesome insertion of a question mark for INPUTing numeric values can be brought under control by using combinations of PRINT and INPUT statements. Consider this sequence of BASIC statements:

```
PRINT "ENTER A NUMBER BETWEEN 0 AND 9"  
INPUT N
```

The response on the screen is:

```
ENTER A NUMBER BETWEEN 0 AND 9  
?
```

The question mark is followed by a blinking cursor symbol, but at least the prompting message appears to be more of a declaration than a question.

Other features of the INPUT statement, including the entry of a series of values in response to a single INPUT line, are well documented in the standard Integer BASIC literature.

**CONTROLLING PROGRAM FLOW WITH INPUT** The fact that an INPUT statement halts the progress of a program until the user strikes the RETURN key makes it a valuable tool for controlling the flow of events within the program. The INPUT can thus represent a critical point in the execution of a program.

**A Common Example: The YES/NO Situation** A lot of BASIC programs execute a relatively long series of operations, and then reach a critical point where the user has to decide whether to repeat the sequence or end it. That generally comes down to responding to a string-type INPUT statement that expects a YES or NO response from the keyboard.

Here is a sequence of statements that appears in a great many Integer BASIC programs:

---

```
100 INPUT "DO YOU WANT TO PLAY AGAIN (Y/N)? ",S$
110 IF S$="Y" THEN 10
120 IF S$="N" THEN END
130 GOTO 100
```

---

The prompting message imbedded in the INPUT statement asks the program user to enter a Y or an N character. If the user responds by entering a Y, the program goes to line 10 (presumably some meaningful entry point for playing the game again); if he enters an N the conditional statement in line 120 is satisfied and the program comes to an END. But if the user happens to enter anything but a simple Y or N, the program defaults to line 130 to repeat the entire INPUT sequence. The routine, in other words, is goof-proofed against erroneous keyboard responses to the prompting message.

The real point of the example, however, is to show how INPUT statements can be used for controlling the flow of events in a program where just one of two possible paths is available. The same idea applies equally well to responses other than Y or N. For instance:

---

```
100 INPUT "DO YOU WANT NORMAL (N) OR FLASHING (F) CHARACTERS?",S$
110 IF S$#"N" THEN 130
120 POKE 50,255: GOTO 150
130 IF S$#"F" THEN 100
140 POKE 50,127
150 REM
```

---

**Program Menus** Program menus offer the user a wide range of choices regarding what is to be done next. There is virtually no limit to the number of items that may be offered in a menu.

Suppose you have written a series of subroutines in a BASIC program that treat two numbers in an arithmetic fashion. The menu task is to give the user an opportunity to select execution of one of the following subroutines:

Line 1000—ADD subroutine  
Line 2000—SUBTRACT subroutine  
Line 3000—MULTIPLY subroutine  
Line 4000—DIVIDE subroutine

Through the following examples of menu operations, we will merely refer to those subroutine line numbers and assume that they are complete, operating subroutines.

---



There are a lot of different ways to format the menu for getting at one of those subroutines. Here is one example:

---

```
100 CALL -936
110 PRINT "SELECT ONE (1,2,3,4 OR 5)"
120 PRINT
130 TAB 5: PRINT "1 -- ADD"
140 TAB 5: PRINT "2 -- SUBTRACT"
150 TAB 5: PRINT "3 -- MULTIPLY"
160 TAB 5: PRINT "4 -- DIVIDE"
170 TAB 5: PRINT "5 -- END THE PROGRAM"
180 PRINT
190 INPUT MS
200 IF MS>=1 AND MS<=4 THEN GOTO MS*1000
210 IF MS=5 THEN END
220 PRINT
230 PRINT "ENTRY ERROR. TRY AGAIN ..."
240 GOTO 110
```

---

Lines 110 through 180 are really little more than an extended, PRINTed prompt message. It tells the user exactly what to do: enter a number from 1 to 5. The INPUT statement in line 190 halts the progress of the program until the user enters a numeric value.

Line 200 tests the values to make certain they fall within the allowed range. If the entered value is a 1, 2, 3, or 4, the GOTO portion of that line sends operations to routines beginning at lines 1000, 2000, 3000, or 4000, respectively. If the user has entered a 5, line 210 detects that fact and ENDS the program.

Program lines 220 through 240 make up a default routine that handles incorrect responses to the INPUT statement. The routine informs the user of the error and returns to line 110 to print the entire menu and give the user a chance to INPUT a proper selection.

The next menu example accomplishes exactly the same task, but uses a different input format. In this example the user is asked to enter a letter instead of a number. The extended prompt message occupies program lines 110 through 170, and the corresponding INPUT statement appears in line 180.

The task of decoding the keyboard inputs is a bit more cumbersome here because each input has to be decoded separately in lines 190 through 230.

If the user fails to enter one of the designated string characters, the program defaults to the error routine in line 250. The error routine repeats the list of valid characters to be entered and returns control to the INPUT statement in line 180.

---

---

```

100 CALL -936
110 PRINT "ENTER YOUR SELECTION AS A,B,C,D OR E:"
120 PRINT
130 TAB 5: PRINT "(A) ADD"
140 TAB 5: PRINT "(B) SUBTRACT"
150 TAB 5: PRINT "(C) MULTIPLY"
160 TAB 5: PRINT "(D) DIVIDE"
170 TAB 5: PRINT "(E) END THE PROGRAM"
180 INPUT MS$
190 IF MS$="A" THEN 1000
200 IF MS$="B" THEN 2000
210 IF MS$="C" THEN 3000
220 IF MS$="D" THEN 4000
230 IF MS$="E" THEN 260
240 PRINT
250 PRINT "PLEASE ENTER A,B,C,D OR E": GOTO 180
260 INPUT "ARE YOU SURE YOU WANT TO QUIT (Y/N)?",S$
270 IF S$="Y" THEN END
280 GOTO 100

```

---

This particular example also asks the user to confirm the end-of-program selection. Upon INPUTing an E, the conditional statement in line 230 gives control to line 260, which PRINTs the prompt message, ARE YOU SURE YOU WANT TO QUIT (Y/N)? The response is entered into S\$, and line 270 ends the program if the response is Y; otherwise everything starts all over from line 100.

A third kind of menu formatting asks the user to enter special symbols that have some direct significance to the operations he or she wishes to select. This next example asks for the arithmetic symbols for addition, sub-

---

```

100 CALL -936
110 PRINT "SELECT AN OPERATION:"
120 PRINT
130 TAB 5: PRINT "ENTER + FOR ADDITION"
140 TAB 5: PRINT "ENTER - FOR SUBTRACTION"
150 TAB 5: PRINT "ENTER * FOR MULTIPLICATION"
160 TAB 5: PRINT "ENTER / FOR DIVISION"
170 TAB 5: PRINT "ENTER Q IF YOU WANT TO QUIT"
180 PRINT
190 INPUT MS$
200 IF MS$="+" THEN 1000
210 IF MS$="-" THEN 2000
220 IF MS$="*" THEN 3000
230 IF MS$="/" THEN 4000
240 IF MS$="Q" THEN END
250 PRINT
260 PRINT "PLEASE ENTER +,-,*, / OR Q."
270 PRINT "TRY AGAIN ..."
280 GOTO 190

```

---

traction, multiplication, and division. It also accepts a Q input for ending the program.

As far as the keyboard environment is concerned, menu routines are little more than simple combinations of the PRINT/INPUT statements described earlier in this chapter. The message portion can be quite involved, but the INPUT variable is still quite simple. Of course the menu is meaningless unless you include an INPUT error-correcting routine and a variable decoding routine that calls the correct subroutine.

**STROBING THE KEYBOARD WITH PEEK STATEMENTS** The INPUT statement of BASIC is not the only mechanism for entering keystroke information during the execution of a program. The Apple system has a place in RAM that is assigned to the keyboard *strobe*. We'll call that location KBD, and its address is -16384.

PEEKing into KBD tells whether or not any key is depressed, and which key it is.

There is a bit more to PEEKing into KBD than simply getting some value out of it, however. Doing a PEEK to KBD, such as in C=PEEK(-16384), not only assigns the current keyboard status to variable C, but latches that value in KBD. KBD thus continues carrying that character code until another instruction calls for PEEKing into KBD. That is not always a desirable situation. It is necessary to reset, or clear, the content of KBD to its no-key-depressed status; and that is done by POKEing a 0 to the keyboard status register, called KBDSTB, at RAM address -16368. Failing to clear KBDSTB can cause a programmer some confusing problems.

A PEEK to keyboard operation involves two separate steps:

PEEK(-16384)—PEEK to KBD and latch the keyboard status in KBD.

POKE -16368,0—Clear the keyboard status to its no-key-depressed condition.

**The Keyboard Character Codes** On executing a PEEK(-16384), the system looks at the content of KBD and moves on to the next instruction. If, during that interval, no key is depressed, KBD will contain a value that is less than 128. But if some key is depressed during the short execution time of that PEEK-to-KBD operation, KBD will take on some value between 128 and 222.

Table 5-1 shows the keystrokes and their respective keyboard character codes, or key codes. Notice that most of the key codes are generated by single key depressions, while a few are the result of depressing the CTRL key and a different key simultaneously.

Suppose, for example, that you want to use the CTRL-A combination for some particular control purpose. What key code does that produce? According to the table, that particular keyboard operation puts a code 129 into KBD.

Unquestionably, a table such as this one is vital for preparing programs that PEEK to KBD to pick up keystroke entries and do something meaningful with them.

Here is a little program that lets you confirm the information supplied in Table 5-1:

---

```
100 CALL -936
110 C= PEEK (-16384)
120 IF C<128 THEN 110
130 POKE -16368,0
140 PRINT C
150 GOTO 110
```

---

On running this program, strike some keys and notice the corresponding key codes printed on the screen. Compare the results with the information supplied in Table 5-1.

An analysis of the program is as follows:

Line 100 homes the cursor and clears the screen.

Line 110 strobes the keyboard by PEEKing into KBD, and assigns the content of KBD to variable C.

Line 120 loops back to strobe the keyboard again if C is less than 128, indicating that no key is depressed.

Line 130 resets the keyboard strobe by POKEing a zero into location KBDSTB.

Line 140 prints the value that is assigned to variable C by line 110.

Line 150 loops back to line 110 to fetch the value of the next keystroke.

Program lines 120 and 130 are especially important to the proper operation of such programs. Delete line 120, for example, and see what happens. Instead of seeing the key codes as you strike various keys, the program runs an endless list of numbers having values less than 128. Those are the values that are loaded into KBD (and assigned to variable C by our program) whenever a key is not being depressed. Depress a key, and you will see the proper value appearing just once in that fast-moving list. Thus, line 120 avoids the printing of a lot of meaningless values less than 128.

Table 5-1. Keyboard Character Codes

Keystroke	Key Code	Keystroke	Key Code	
@	192	3	179	
A	193	4	180	
B	194	5	181	
C	195	6	182	
D	196	7	183	
E	197	8	184	
F	198	9	185	
G	199	:	186	
H	200	;	187	
I	201	<	188	
J	202	=	189	
K	203	>	190	
L	204	?	191	
M	205	←	136	
N	206	RETURN	141	
O	207	→	149	
P	208	ESC	155	
Q	209	^	222	
R	210	CTRL-@	128	(Same as CTRL-H)
S	211	CTRL-A	129	(Same as CTRL-M)
T	212	CTRL-B	130	(Same as CTRL-U)
U	213	CTRL-C	131	
V	214	CTRL-D	132	
W	215	CTRL-E	133	
X	216	CTRL-F	134	
Y	217	CTRL-G	135	
Z	218	CTRL-H	136	
space	160	CTRL-I	137	
!	161	CTRL-J	138	
"	162	CTRL-K	139	
#	163	CTRL-L	140	
\$	164	CTRL-M	141	
%	165	CTRL-N	142	(Same as ←)
&	166	CTRL-O	143	
	167	CTRL-P	144	
(	168	CTRL-Q	145	
)	169	CTRL-R	146	
*	170	CTRL-S	147	(Same as RETURN)
+	171	CTRL-T	148	
'	172	CTRL-U	149	(Same as →)
-	173	CTRL-V	150	
•	174	CTRL-W	151	
/	175	CTRL-X	152	
0	176	CTRL-Y	153	
1	177	CTRL-Z	154	
2	178			

Insert line 120 back into the program, and the routine will loop rapidly between lines 110 and 120 until you strike a key.

Line 130 is the one that resets the keyboard strobe. Delete that line and run the program again. You will see a long string of numbers again. This time, however, striking a key causes the appropriate key code to appear in the list; in fact it doesn't go away until you strike another key. Indeed, PEEKing to KBD sets a flip-flop, or latch, function in the keyboard system. Clearing, or resetting, that latch is a matter of doing a POKE-0 to KBDSTB. Insert line 130 back into the program, and you'll find everything working nicely once again.

If you compare the table of keystroke codes with the video text codes in Table 4-5, you will find that many of them share the same characters and codes. The following program lets you play around with keystrokes and the text characters they might create on the screen:

---

```
100 CALL -936
110 C= PEEK (-16384)
120 IF C<128 THEN 110
130 POKE -16368,0
140 PRINT C;
150 POKE 256* PEEK (41)+ PEEK (40)+ PEEK (36)+4,C
160 CALL -926
170 GOTO 110
```

---

The program prints both the key codes and a corresponding video text character. It strobcs the keyboard to pick up a valid key code, assigns it to variable C, and resets the keyboard strobe as in the previous program. Line 140 prints the key code as before, but the semicolon suppresses the normal linefeed and carriage return operation. Line 150 then comes up with the current cursor position, adds four spaces, and then POKES the character code to the screen. POKEing the key code to the screen causes the video system to print the character it represents. PRINTing to the screen, as in line 140, merely prints the key code. There is an important distinction between PRINTing a key code to the screen and POKEing it there.

**An Example: Printing a Lot of Text** When working from the Apple monitor, the BASIC immediate command mode, or with an INPUT statement, you may enter no more than 256 consecutive characters. That is a limitation imposed by the size of the GETLN input buffer that is described later in this book. Directly strobing the keyboard and POKEing the characters to the video text RAM offers a chance to type in a string of messages of an indefinite length.

The following program POKes characters to the video system as they are entered at the keyboard. In a sense, it represents the beginning of a primitive word processor.

---

```
100 CALL -936
110 C= PEEK (-16384)
120 IF C<128 THEN 110
130 POKE -16368,0
140 POKE 256* PEEK (41)+ PEEK (40)+ PEEK (36),C
150 CALL -1036
160 GOTO 110
```

---

Lines 110 through 130 strobe the keyboard, looking for a valid key-stroke. Upon finding one, it assigns the key code to variable C and resets the keyboard strobe. Line 140 POKes the character to the current cursor address location, and line 150 advances the cursor to the next location.

You can type in a full screen of text, and the screen will scroll upward in the normal fashion as you continue entering characters “below” the last line. You can, in fact, set up a smaller text window (see Chapter 2) and work the characters into that space on the screen.

From a word-processing viewpoint, however, the simple program suffers from some serious problems. For one, it has no editing features; you cannot backspace and erase errors, for instance. The program doesn’t show the cursor symbol, either. So, if you do a series of spaces, you might not be sure where the next character will appear on the screen.

Nevertheless, the program illustrates one kind of application of the keyboard strobe feature of the Apple system. Later discussions in this chapter show how to do some elementary editing.

**SINGLE-KEYSTROKE CONTROL OF A PROGRAM**      The notion of strobing the keyboard by PEEKing into KBD offers some fine techniques for controlling the flow of a program. This method rivals, and often surpasses, the INPUT-statement control method.

**Resuming Stopped Operations**      A lot of different kinds of text and graphics programs call for halting the ongoing operations until the user has a chance to view conditions on the screen. The user is then given the option of striking a key to resume operations on the screen.

The following is an example of a program that uses PEEK to resume operations upon the striking of *any* key:

---

```
100 N=0
110 L=0: CALL -936
120 PRINT N
130 N=N+1:L=L+1
140 IF L<20 THEN 120
150 PRINT : TAB 8
160 PRINT "STRIKE ANY KEY TO CONTINUE..."
170 IF PEEK (-16384)<128 THEN 170
180 POKE -16384,0
190 GOTO 110
```

---

(You will understand the following short analysis better if you first enter and run the program.)

The conditional statement in program line 140 is responsible for halting the progress of the counting operation. Lines 150 and 160 then format and print the prompting message.

Lines 170 through 190 are the ones most important to the current discussion. Line 170 strobes the keyboard and compares the content of KBD with the value of 128. If the content of KBD is less than 128, it means that the user has not yet made a keystroke in response to the prompting message. As long as that condition exists, the program “buzzes” on line 170.

The moment the user strikes any key, the value in KBD becomes 128 or greater, and the program goes to line 180 to reset the keyboard latch, and then to line 110 to resume the printing of successively larger numbers.

A BASIC statement of the form shown here in line 170 can always be used to halt the flow of a program until the user strikes any key on the keyboard. But you can accomplish the same thing with an INPUT statement:

```
160 INPUT "STRIKE RETURN KEY TO CONTINUE ...",$$:GOTO 110
```

That one line replaces lines 160 through 190 in the PEEK-to-KBD version.

The only advantage of the PEEK-to-KBD version of this particular application is that it allows the user to resume operations by striking *any* key. Other situations, however, offer far more compelling advantages. Consider the application described next.

**Stopping Ongoing Operations**      The following program is the inverse of the one just described. Instead of using a single keystroke to resume operations, it uses a single keystroke to stop them.

Enter and run the program, striking any key to stop the counting operations. This particular version responds to the keystroke by going into a loop, so you will have to do a CTRL-C to get out of the loop and RUN again.



---

```
100 CALL -936
110 TAB 10: POKE 50,63
120 PRINT "STRIKE ANY KEY TO STOP..."
130 POKE 50,255: POKE 34,1
140 N=0
150 PRINT N
160 N=N+1
170 IF PEEK (-16384)<128 THEN 150
180 POKE -16368,0
190 TEXT
200 TAB 1: VTAB 1: CALL -868
210 GOTO 210
```

---

The critical part of the program resides in lines 150 through 180. Lines 150 and 160 print the current value of variable N and increment it by 1. Line 170 then PEEKs to KBD, and if the value is less than 128 (no key depressed), the program loops back to line 150. The program thus cycles continuously through lines 150, 160, and 170 until the user strikes a key.

When the PEEK-to-KBD statement in line 170 detects that a keystroke has indeed occurred, the program goes to line 180 to reset the keyboard latch and bring the program to a conclusion through the remaining lines.

Notice especially that the PEEK-to-KBD operation is a normal part of the program's running cycle. There is no interruption of the program as long as no key is depressed. That cannot be done with an INPUT statement; INPUT statements *always* interrupt the flow of a program. Therein lies the real usefulness of PEEK-to-KBD operations.

The remaining lines in that program are merely window dressing, but we will study them closely for the sake of reviewing some of the text-formatting techniques described earlier in this book.

Line 100 homes the cursor and clears the entire screen.

Line 100 performs a horizontal TAB and sets the system for printing characters in inverse format.

Line 120 prints the prompting message (at TAB 10 on the top line and using inverse text).

Line 130 returns the system to the normal format at the top of the next window at row 1. Setting the top of the window at that point prevents the prompting message from scrolling off the screen during the number-printing operations that follow.

Line 140 initializes the counting variable N.

Lines 150 through 180 print and increment the value of N until a keystroke occurs.

Line 190 restores the normal text window.

Line 200 homes the cursor without disturbing the current text, and then clears the current line to get rid of the prompting message.

Line 210 loops to this line until the user interrupts with a CTRL-C or RESET.

**Toggleing the Operations** If a PEEK-to-KBD statement can be used for resuming program operations, and if it can also be used for stopping operations, then it can be used for toggling the operations on and off. A keystroke at one time can start an operation and a subsequent keystroke can stop it.

Here is the same counting routine used in the previous examples. This time, however, it includes two different kinds of PEEK-to-KBD operations—one to start counting, and one to stop counting.

---

```
100 CALL -936
110 N=0
120 IF PEEK (-16384)<128 THEN 120
130 POKE -16368,0
140 PRINT N
150 N=N+1
160 IF PEEK (-16384)<128 THEN 140
170 POKE -16368,0
180 GOTO 120
```

---

The conditional statement in line 120 keeps the counting operation stopped until the user strikes any key. That program line “buzzes” to itself as long as it sees no key depression. In effect, it *starts* the program operations as soon as the user strikes any key.

The statement in line 160 is quite similar to that in line 120; but as long as this one is satisfied (as long as the content of KBD is less than 128), the printing and counting operations take place. Line 160 is responsible for detecting the keystroke that will *stop* the counting operation. As soon as that keystroke occurs, line 170 clears the keyboard strobe, and line 180 returns the program to line 120.

Enter and run the program. You will find that the first keystroke starts the counting operation and that the next keystroke stops it. Resume the counting by striking any key again. You can thus toggle this counting operation on and off any number of times. Getting out of the whole program is a matter of performing a CTRL-C or RESET.

I have omitted the prompting messages from this example so that the real reason for presenting it will stand out more clearly. It is a bare-bones version of a toggling program. The following listing uses the same toggling mechanisms, but includes the prompting messages as well.

Incidentally, I generated this expanded version from the original one

by inserting the prompting and formatting statements, and then using the Integer BASIC RENUM feature to get the line numbers back into an orderly form.

---

```
100 CALL -936
110 N=0
120 TAB 10: VTAB 1
130 POKE 50,63
140 PRINT "STRIKE ANY KEY TO START"
150 POKE 50,255: POKE 34,1
160 IF PEEK (-16384)<128 THEN 160
170 POKE -16368,0
180 TAB 1: VTAB 1: CALL -868
190 TAB 10: POKE 50,63
200 PRINT "STRIKE ANY KEY TO STOP"
210 POKE 50,255
220 PRINT N
230 N=N+1
240 IF PEEK (-16384)<128 THEN 220
250 POKE -16368,0
260 GOTO 120
```

---

The following is a line-by-line analysis of the program:

- Line 100 homes the cursor and clears the screen.
- Line 110 initializes the counting variable.
- Line 120 tabs the first prompting message.
- Line 130 sets the inverse character format.
- Line 140 prints STRIKE ANY KEY TO START (at TAB 10 on the top line, using inverse characters).
- Line 150 returns to normal format, and sets the top of the text window to row 1. (That prevents the subsequent operations from scrolling the prompting message off the top of the screen.)
- Line 160 holds up operations until a keystroke occurs.
- Line 170 clears the keyboard strobe.
- Line 180 sets the cursor to the beginning of the top line on the screen, and clears to the end of that line. The purpose of this line is to erase the current prompting message.
- Line 190 tabs the next message on the top line, and sets the inverse format.
- Line 200 prints STRIKE ANY KEY TO STOP.
- Line 210 sets the normal format.
- Lines 220 through 240 print and increment the value of N until a keystroke occurs.
- Line 250 clears the keyboard strobe.
- Line 260 goes back to line 120 to do the restart routine.

**A Full Program Example** Listing 5-1 represents a full, working program that uses some single-keystroke operations for control purposes. It is a reaction-time tester. It displays a square of white light on the screen at some random time after you begin a delay cycle. From the moment the square appears, the program shows the elapsed time in seconds, tenths of seconds and hundredths of seconds. The timer stops as soon as you strike any key.

Enter the program and play with it for a while before you read the following analysis. You will find the program incorporates some PEEK-to-KBD techniques as well as some special text-formatting procedures described in earlier chapters.

Line 100 sets the normal format, homes the cursor, and clears the entire screen.

Lines 110 through 140 tab and print a program title message and a prompting message.

Line 150 holds up further execution of the program until the user strikes any key.

Line 160 clears the keyboard strobe.

Lines 170 through 210 provide animation by scrolling the current messages up one line at a time until they disappear from the screen.

Lines 220 through 330 print an extensive set of instructions (using the normal format set in line 100).

Lines 340 through 370 tab the next message, set the inverse format, print the prompting message **STRIKE ANY KEY TO BEGIN THE DELAY CYCLE**, and wait for a keystroke.

Line 380 clears the keyboard strobe.

Line 390 homes the cursor and clears the screen.

Lines 400 through 420 perform an interruptible time delay of random duration. Line 410 makes it possible to interrupt this timing sequence by striking any key. If no keystroke occurs during this timing sequence, the program goes to line 430 and then to 480. But if a keystroke does occur, as sensed by the PEEK-to-KBD instruction in line 410, the timing sequence is aborted, and program control goes to line 440.

Lines 440 through 470 are called whenever the user strikes a key before the white square appears on the screen. The routine clears the keyboard strobe (originally set by striking a key at line 410), sets the flashing format, and prints a message at the beginning of the top line on the screen. (The cursor is set to that point by line 390.) Line 470 returns the program to line 340, where the user is prompted to start the delay cycle all over again.

## Listing 5-1. Reaction Time Tester.

---

```
100 POKE 50,255: CALL -936
110 VTAB 8: TAB 9
120 PRINT "*** REACTION TIME TESTER ***"
130 VTAB 16: TAB 1
140 PRINT "STRIKE ANY KEY TO START..."
150 IF PEEK (-16384)<128 THEN 150
160 POKE -16368,0
170 VTAB 24
180 FOR N=0 TO 15
190 PRINT
200 FOR T=0 TO 100: NEXT T
210 NEXT N
220 CALL -936
230 PRINT "AFTER STARTING THE DELAY CYCLE, LOOK"
240 PRINT "FOR A WHITE SQUARE THAT WILL"
250 PRINT "APPEAR IN THE UPPER LEFT-HAND"
260 PRINT "CORNER OF THE SCREEN. RESPOND"
270 PRINT "BY STRIKING ANY KEY AS SOON AS"
280 PRINT "POSSIBLE. YOUR REACTION TIME"
290 PRINT "WILL THEN APPEAR IN THE UPPER"
300 PRINT "RIGHT-HAND CORNER."
310 PRINT
320 PRINT "DO NOT STRIKE THE KEY TOO SOON."
330 PRINT "THAT ABORTS THE CYCLE."
340 TAB 1: VTAB 16
350 POKE 50,63
360 PRINT "STRIKE ANY KEY TO BEGIN THE DELAY CYCLE."
370 IF PEEK (-16384)<128 THEN 370
380 POKE -16368,0
390 CALL -936
400 FOR T=0 TO 1000+ RND (4000)
410 IF PEEK (-16384)>127 THEN 440
420 NEXT T
430 GOTO 480
440 POKE -16368,0
450 POKE 50,127
460 PRINT "TOO SOON. CALM DOWN."
470 GOTO 340
480 NH=48:NT=48:NU=48
490 POKE 1024,32
500 POKE 1054,NU
510 POKE 1055,46
520 POKE 1056,NT
530 POKE 1057,NH
540 IF PEEK (-16384)>127 THEN 620
550 NH=NH+1
560 IF NH<58 THEN 490
570 NH=48:NT=NT+1
580 IF NT<58 THEN 490
590 NT=48:NU=NU+1
600 IF NU>57 THEN 660
610 GOTO 490
620 POKE -16368,0
630 VTAB 1: TAB 1
640 PRINT "YOUR REACTION TIME:"
650 GOTO 340

660 VTAB 1: TAB 1
670 POKE 50,127
680 PRINT "YOU TOOK TOO LONG. FORGET IT."
690 GOTO 340
```

Lines 480 through 610, discussed next, make up the program's elapsed-time counting loop. Like the delay time sequence, this loop is interruptible by means of the PEEK-to-KBD operation.

Line 480 initializes the three digits in the elapsed-time counter. NH carries hundredths of seconds, NT carries tenths of seconds, and NU carries full seconds. They are initialized to 48 instead of to 0 because later operations POKE those values to the screen. Recall that you must POKE character codes, and not the values themselves, to video memory. Character code 48 represents an inverse 0.

Line 490 POKEs the square of light to the screen. This is the signal that the user is to strike a key.

Line 500 POKEs the current full-seconds character to the screen.

Line 510 POKEs an inverse period to the screen.

Lines 520 and 530 POKE tenths and hundredths of seconds to the screen.

Line 540 performs a PEEK-to-KBD. If a key is depressed, control is given to line 620; otherwise, the elapsed-time counter is incremented.

Lines 550 through 610 run the elapsed-time counter. If the time exceeds 9.99 seconds, as sensed by the conditional statement in line 600, then the counting sequence is ended by going to line 660.

Lines 620 through 650 are executed if the user makes a proper response to the beginning of the elapsed-time sequence. They reset the keyboard strobe, format and print YOUR REACTION TIME, and return control to line 340 to give the user a chance to begin a new delay cycle.

Lines 660 through 690 are executed if the user allows the elapsed time to run past 9.99 seconds. The routine prints YOU TOOK TOO LONG in flashing characters, and then loops the program back to line 340.

The main points, in the context of this section on PEEK-to-KBD control, are the interruptible timing sequences. The program runs along one line of operations if no keystroke occurs, but enters a different routine if a keystroke does occur. Such operations cannot be duplicated using INPUT statements.

## DECODING SINGLE KEYSTROKES FOR CONTROL PURPOSES

The single-keystroke control techniques described in the previous section are adequate for initiating a program sequence, halting or interrupting a sequence, or toggling between two different program sequences. As long as there are no more than two control options—no more than two events that can occur as the result of doing a PEEK-to-KBD—it makes no difference which key the user strikes. But of course there are many instances where the user must be given the option of selecting two or more routes in

a program. That means it is necessary to define certain keystrokes for certain purposes and decode the key codes as they arrive from the keyboard.

**The YES/NO Situation Revisited**      An earlier topic in this chapter demonstrated how it is possible to use INPUT statements to make a YES or NO decision from the keyboard. Here is an example:

---

```
100 CALL -936
110 FOR N=0 TO 9
120 PRINT N
130 FOR T=0 TO 100: NEXT T
140 NEXT N
150 PRINT
160 PRINT "WANT TO DO THIS THING AGAIN (Y/N) ?"
170 INPUT C$
180 IF C$="Y" THEN 100
190 IF C$="N" THEN END
200 GOTO 170
```

---

The main part of the program occupies lines 100 through 140. It simply clears the screen and prints numerals 0 through 9, with a short time delay inserted between each printing. The control portion uses the remainder of the program.

Line 150 leaves a blank line, and then line 160 asks the user, WANT TO DO THIS AGAIN? Line 170 uses an INPUT statement to assign the user's response to variable C\$. If the response is a Y, then control is picked up from line 100 again. If the response is N, the program comes to an end. Any other keyboard entry is handled by line 200, which repeats the INPUT statement until the user gets it right.

There is nothing technically wrong with that INPUT control scheme. The user strikes the Y or N key and enters that choice by striking the RETURN key as well.

Now, compare that with this version of the same general program that uses a decoded PEEK-to-KBD control instead of INPUT.

---

```
100 CALL -936
110 FOR N=0 TO 9
120 PRINT N
130 FOR T=0 TO 100: NEXT T
140 NEXT N
150 PRINT
160 PRINT "WANT TO DO THIS THING AGAIN (Y/N) ?"
170 C= PEEK (-16384): IF C<128 THEN 170
180 POKE -16368,0
190 IF C=217 THEN 100
200 IF C=206 THEN END
210 GOTO 170
```

---

The two program listings are identical down to line 170. In the latter version, line 170 PEEKS to the keyboard and continues doing so until it detects a keystroke. Then lines 190 and 200 decode the result. If the user strikes the Y key, variable C will be equal to 217. Line 190 handles that situation by sending program control back to line 100 to repeat the counting sequence. If the user responds to the prompting message by striking the N key, the value of C becomes 206, and line 200 responds to that value by bringing the program to an end. Finally, if the keystroke is neither an N nor a Y, the program defaults to line 210 which, in turn, sends program control back to line 170 to give the user another chance.

Line 180 clears the keyboard strobe latch immediately after any sort of keystroke.

Both versions of the program do exactly the same overall task. But if you enter and run both of them, you will find that the PEEK-to-KBD version seems to “feel” better from the user’s point of view. There is an immediate response to a Y or N keystroke, so there is no need to strike the RETURN key at all. If you happen to strike any other key, nothing happens.

A careful application of PEEK-to-KBD routines can transform a mediocre program into a much more pleasing and exciting one.

**The Menu Situation Revisited** “Program Menus” (page 90) described the application of INPUT statements in program menu routines. The idea was to select one of any number of possible paths through a program by INPUTting a numeral, letter, or symbol in response to a multiple-choice menu listing. Using an INPUT statement means that the user must make at least two keystrokes to satisfy the menu routine: one to select the menu item and a second (a RETURN keystroke) to complete the execution of the INPUT statement.

As shown in this section, using a PEEK-to-KBD routine with a menu reduces the number of keystrokes to just one—the one that selects the menu item.

Enter and run Listing 5-2, and see if you agree that you get a positive feeling of immediate interaction with it.

Here is a line-by-line analysis of the menu-selection and keystroke-decoding portion of that program:

Line 160 prints the menu prompting message.

Lines 170 and 180 set the normal format and leave a blank line.

Lines 190 through 250 print the menu listing and leave a blank line.

Lines 260 and 270 PEEK to KBD and assign the current value to variable

F. If there is no keystroke representing the key codes for numerals 1 through 6, the program goes back to PEEK at KBD again. The pro-



## Listing 5-2. Adder, Subtractor, Multiplier.

---

```
100 CALL -936
110 PRINT "ENTER TWO NUMBERS AS A,B"
120 PRINT : PRINT "(-100 THRU +100)"
130 PRINT
140 INPUT A,B
150 CALL -936
160 PRINT "SELECT A FUNCTION (1,2,3,4,5 OR 6):"
170 POKE 50,255
180 PRINT
190 TAB 5: PRINT "1 -- A+B"
200 TAB 5: PRINT "2 -- A-B"
210 TAB 5: PRINT "3 -- B-A"
220 TAB 5: PRINT "4 -- A*B"
230 TAB 5: PRINT "5 -- SELECT NEW NUMBERS"
240 TAB 5: PRINT "6 -- QUIT THE PROGRAM"
250 PRINT
260 F= PEEK (-16384)
270 IF F<177 OR F>182 THEN 260
280 POKE -16368,0
290 CALL -868
300 IF F=177 THEN 360
310 IF F=178 THEN 380
320 IF F=179 THEN 400
330 IF F=180 THEN 420
340 IF F=181 THEN 100
350 END
360 PRINT A;"+";B;"=";A+B
370 GOTO 430
380 PRINT A;"-";B;"=";A-B
390 GOTO 430
400 PRINT B;"-";A;"=";B-A
410 GOTO 430
420 PRINT A;"*";B;"=";A*B
430 PRINT
440 PRINT "CURRENT NUMBERS:"
450 TAB 5: PRINT "A=";A
460 TAB 5: PRINT "B=";B
470 POKE 50,63
480 VTAB 1: TAB 1
490 GOTO 160
```

---

gram “buzzes” on these two lines until the user strikes one of the six numeral keys designated in the menu listing.

Lines 280 and 290 clear the keyboard latch and clear to the end of the current line of text. The program reaches this point only after the user strikes one of the six keys in the menu listing. The CLREOL operation simply clears the summary of the previous arithmetic operation from the screen.

Lines 300 through 350 decode the legitimate keystroke and take the appropriate action.

The entire analysis is almost meaningless if you haven't realized that the key code for numeral 1 is 177, 2 is 178, 3 is 179, and so on. (See Table 5-1.)

Thus far, most of the suggested applications of PEEK-to-KBD routines are nice substitutes for the somewhat more awkward INPUT routines. The next section in this chapter deals with an application that is totally foreign to INPUT techniques.

**An Improved Text Editor** Earlier in this chapter we showed you a primitive form of word processor, or text editor. The little program simply POKEd to the screen any valid, printable character typed at the keyboard. One of its real shortcomings was that the user could not alter any of the text once it was committed to the screen. Now we are in a position to change all that.

Listing 5-3 allows you to type in text and carry out some common editing functions as well. The purpose of the demonstration is to show how it is possible to use PEEK-to-KBD operations for both generating text and setting up control functions.

Enter and run that program. Type in some normal alphanumeric characters and punctuation marks to convince yourself that the program can indeed transfer such keystrokes to the video text system. Then try these control functions:

CTRL-S homes the cursor without erasing any text.

CTRL-T causes an upward linefeed without erasing any text.

CTRL-D causes a downward linefeed without erasing any text.

CTRL-R advances the cursor without erasing any text.

CTRL-L backspaces the cursor without erasing any text.

The symbol → advances the cursor and erases current text.

The symbol ← backspaces the cursor and erases current text.

CTRL-X erases all text from the cursor to the end of the page.

CTRL-F erases all text from the cursor to the end of the current line.

RETURN causes a linefeed and carriage return.

You will also see that the program displays a text cursor. The more primitive version of the program did not do that.

Here is an analysis of Listing 5-3:

Line 110 fetches the current cursor position and assigns it to variable CP.

Line 120 PEEKs at the current character, calculates the character code for its inverse or flashing format, and POKEs it to the current cursor position. This line is responsible for generating the program's cursor symbol.

Lines 130 and 140 PEEK to KBD. If no key is depressed, they PEEK to KBD again.

### Listing 5-3. Improved Text Editor.

---

```
100 CALL -936
110 CP=256* PEEK (41)+ PEEK (40)+ PEEK (36)
120 POKE CP, PEEK (CP)-128
130 K= PEEK (-16384)
140 IF K<128 THEN 130
150 POKE -16368,0
160 IF K<160 THEN 200
170 POKE CP,K
180 CALL -1036
190 GOTO 110
200 POKE CP, PEEK (CP)+128
210 IF K=132 THEN 1320
220 IF K=134 THEN 1340
230 IF K=136 THEN 1360
240 IF K=140 THEN 1400
250 IF K=141 THEN 1410
260 IF K=146 THEN 1460
270 IF K=147 THEN 1470
280 IF K=148 THEN 1480
290 IF K=149 THEN 1490
300 IF K=152 THEN 1520
310 GOTO 130
1320 CALL -922: GOTO 110
1340 CALL -868: GOTO 110
1360 POKE CP,160: GOTO 1400
1400 CALL -1008: GOTO 110
1410 CALL -926: GOTO 110
1460 CALL -1036: GOTO 110
1470 VTAB 1: TAB 1: GOTO 110
1480 CALL -998: GOTO 110
1490 POKE CP,160: GOTO 1460
1520 CALL -958: GOTO 110
```

---

Line 150 clears the keyboard strobe.

Line 160 jumps to line 200 to execute the control operation if the keystroke represents a control character.

Line 170 prints the character typed if it wasn't a control character.

Line 180 advances the cursor.

Line 190 returns to get the next keystroke.

Line 200 replaces the cursor symbol with the normal format version of the current character.

Lines 210 through 310 decode the control keystroke and send operations to the appropriate lines. Specifically:

Line 210 decodes CTRL-D.

Line 220 decodes CTRL-F.

Line 230 decodes left arrow.  
Line 240 decodes CTRL-L.  
Line 250 decodes RETURN.  
Line 260 decodes CTRL-R.  
Line 270 decodes CTRL-S.  
Line 280 decodes CTRL-T.  
Line 290 decodes the right arrow key.  
Line 300 returns to line 130 to get the next keystroke if the keystroke is not a proper control character.  
Line 1320 performs a downward linefeed (CTRL-D) and gets the next keystroke.  
Line 1340 erases to end of line (CTRL-F) and gets the next keystroke.  
Line 1360 erases the current character and jumps to line 1400 to backspace (left arrow).  
Line 1400 backspaces (CTRL-L) and gets the next keystroke.  
Line 1410 performs a linefeed and carriage return (RETURN) and gets the next keystroke.  
Line 1460 advances the cursor (CTRL-R) and gets the next keystroke.  
Line 1470 homes the cursor (CTRL-S) and gets the next keystroke.  
Line 1480 performs an upward linefeed (CTRL-T) and gets the next keystroke.  
Line 1490 erases the current character and jumps to line 1460 to advance the cursor (right arrow).  
Line 1520 erases to the end of the page (CTRL-X) and gets the next keystroke.

All of the CALL statements take advantage of the cursor-related routines cited in Chapter 2.

# The Low-Resolution Graphics Environment

## 6

One of the truly appealing features of the Apple computer system is its versatile and colorful low-resolution graphics mode of operation. Properly used, this scheme can provide interesting, entertaining, and useful color graphics functions.

The low-resolution graphics screen shares video RAM addresses with the text screen. The two screens overlap exactly, so many of the special text techniques described in an earlier chapter can be applied equally well to the low-resolution graphics mode.

**THE ELEMENTARY PRINCIPLES** Integer BASIC includes a small group of statements that are especially designed to simplify the programming of low-resolution graphics routines. Even if you have already mastered these principles, you might do well to follow this summary closely; the ideas are the framework for more detailed discussions later in this chapter.

**The GR and TEXT Statements** Integer BASIC's GR and TEXT statements are intended to switch the Apple system between its low-resolution and normal text modes. Doing a GR, either from the keyboard in the command mode or within a BASIC program, sets the video system for displaying normal, low-resolution graphics. Doing a TEXT, either from the immediate command mode or within a program, returns the video system to normal text.

The *normal* low-resolution graphics mode is one that is characterized by a mixture of graphics and text locations on the screen. The text area occupies the four lower lines (text rows 20 through 23), and the graphics area fills the remainder of the screen.

The graphics area is divided into 40 rows of 40 graphics locations. That figures out to 1600 separate places on the screen where you can plot small

rectangles of color. The 40 columns in each row are addressed just as text character locations are addressed—0 through 39. Likewise, the 40 rows of low-resolution graphics locations can be labeled 0 through 39. (See the normal graphics scheme illustrated in Fig. 6-1.)

You ought to be able to appreciate the fact that there are 40 rows of graphics locations occupying the space that is normally allotted to 20 rows of text. In other words, there are twice as many graphics rows in that screen area than text rows.

Executing a GR statement sets up the normal graphics mode, clears the low-resolution graphics portion of the screen, but does not affect the text area. It is not uncommon, then, to see a GR statement in a program immediately followed by a CALL -936. The latter statement clears the four-line text window and homes the cursor within it.

Executing a TEXT statement simply returns the system to the full text mode (40 columns, 24 rows) and does not clear anything. That is why you often see a lot of seemingly meaningless text in the upper portion of the screen after doing a TEXT statement to get out of the low-resolution graphics mode. For that reason, the TEXT statement is frequently followed by a CALL -936 to home the cursor and clear the screen.

**The COLOR and PLOT Statements** The COLOR statement determines the color to be plotted on the screen. The COLOR codes and their corresponding colors are shown in Table 6-1.

The syntax for the COLOR statement is:

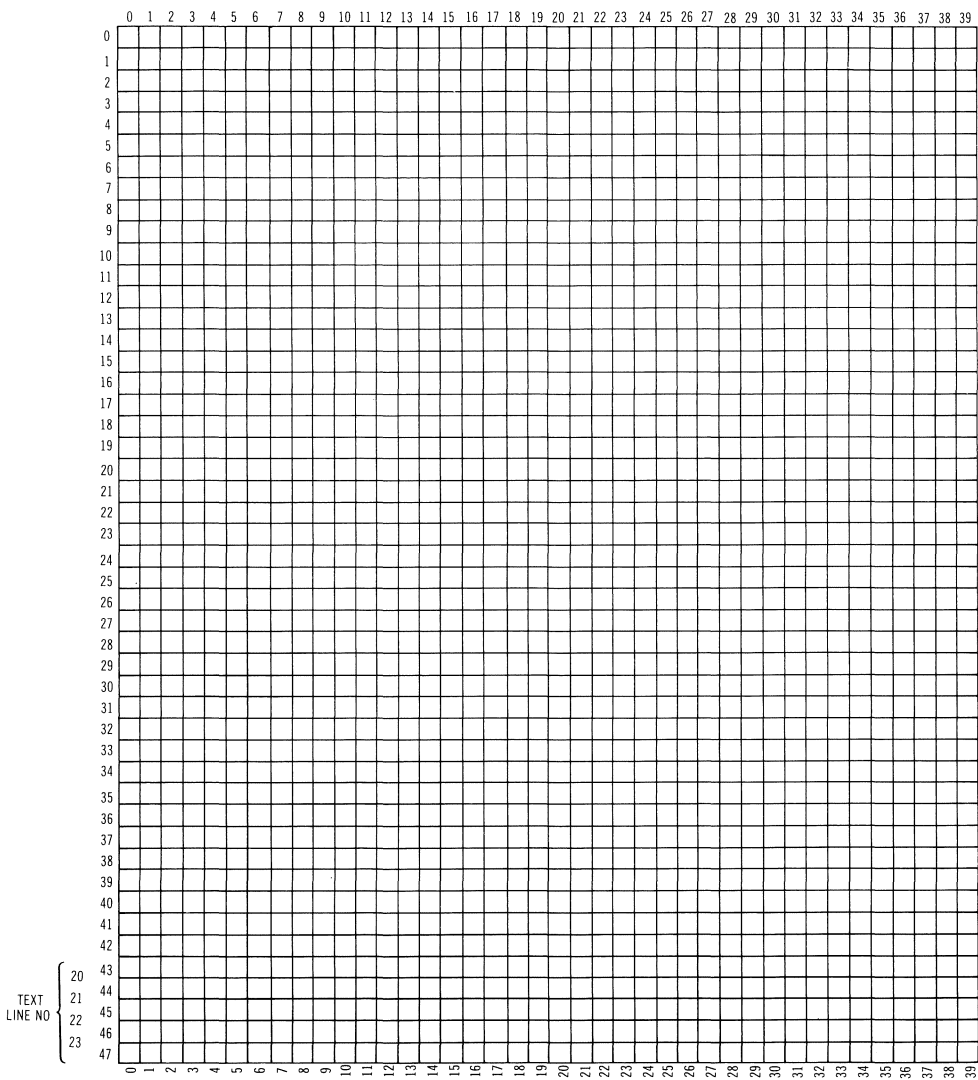
COLOR=c

where c is the desired color code from Table 6-1. COLOR must precede the PLOT statement, described next.

The PLOT statement plots the designated color to a screen location determined according to the special row-and-column addressing format. The syntax for the PLOT statement is:

PLOT x,y

where x is the desired column (0–39) and y is the low-resolution graphics row (0–39).



**Fig. 6-1. Normal graphics scheme.**

**Table 6-1. Color Codes**

<b>COLOR Code</b>	<b>Color</b>
0	Black
1	Magenta
2	Dark Blue
3	Purple
4	Dark Green
5	Grey 1
6	Medium Blue
7	Light Blue
8	Brown
9	Orange
10	Grey 2
11	Pink
12	Light Green
13	Yellow
14	Aquamarine
15	White

Consider the following program:

---

```
10 GR
20 CALL -936
30 COLOR=9
40 PLOT 20,20
50 COLOR=13
60 PLOT 0,0
70 END
```

---

Lines 10 and 20 in that program set up the normal low-resolution graphics mode and clear the text portion of the screen. Those two lines, in effect, clear the entire screen because the execution of GR always clears the graphics area.

Lines 30 and 40 work together to plot an orange rectangle near the middle of the screen. Line 30 sets up the color according to Table 6-1, and line 40 plots a rectangle of that color at column 20, graphics row 20. Then lines 50 and 60 plot a yellow rectangle in the extreme upper left-hand corner of the screen. Again, line 50 sets the color and line 60 plots that color at the designated coordinates—0,0 in this case.

---



Line 70 ends the program. You will find that the BASIC prompt symbol then appears in the text area at the bottom of the screen.

You can, of course, write programs that plot a number of low-resolution color blocks in sequence. Consider this idea:

---

```
10 GR
20 CALL -936
30 COLOR=3
40 FOR X=0 TO 39
50 PLOT X,0
60 NEXT X
70 END
```

---

This program plots a purple line across the top of the screen. Technically speaking, it plots a series of 40 purple rectangles.

Enter the next program into your Apple, RUN it, and observe its behavior. See if you can explain to yourself how it works.

---

```
100 GR
110 CALL -936
120 COLOR=4
130 FOR X=0 TO 39: FOR Y=0 TO 39
140 PLOT X,Y
150 NEXT Y: NEXT X
160 COLOR=15
170 FOR X=0 TO 39
180 PLOT X,0: PLOT X,39
190 NEXT X
200 FOR Y=0 TO 39
210 PLOT 0,Y: PLOT 39,Y
220 NEXT Y
230 END
```

---

**The HLIN and VLIN Statements** Integer BASIC includes two statements that are intended to simplify the programming task of drawing straight horizontal or vertical lines on the low-resolution graphics screen. HLIN draws horizontal lines at designated row positions, and VLIN draws vertical lines at designated column positions.

The form of HLIN is:

HLIN x1,x2 AT y

This statement draws a horizontal line of a previously determined color between columns x1 and x2 at graphics row y. The range of values

---

for all three terms must be from 0 to 39, and x2 must be greater than x1 (HLIN always draws from left to right.)

The form of VLIN is:

VLIN y1,y2 AT x

This statement draws a vertical line between graphics rows y1 and y2, and at column x. Again, the range of values must be from 0 to 39. Since VLIN always draws from top to bottom, y2 must be greater than y1.

Try this demonstration program:

---

```
10 GR
20 CALL -936
30 COLOR=7
40 HLIN 0,39 AT 20
50 VLIN 0,39 AT 20
60 END
```

---

The program draws a set of light blue horizontal and vertical lines that intersect near the middle of the screen. Notice that the vertical line is wider than its horizontal counterpart. That is not an effect created by the HLIN and VLIN statements. Rather, it illustrates the fact that the low-resolution graphics scheme uses rectangular blocks of color—blocks that are about twice as wide as they are high.

If you would like to give those two lines the same width, you must double the width of the horizontal line. There is no way to reduce the width of the vertical line. So try this:

---

```
10 GR
20 CALL -936
30 COLOR=7
35 FOR N=0 TO 1
40 HLIN 0,39 AT 20+N
45 NEXT N
50 VLIN 0,39 AT 20
60 END
```

---

Lines 35 through 45 draw the horizontal line twice—once at graphics row 20 and then again at row 21.

Enter, RUN, and observe the action of the following program:

---

```
100 GR
110 CALL -936
120 COLOR=4
130 FOR X=0 TO 39
140 VLIN 0,39 AT X
150 NEXT X
160 COLOR=15
170 FOR Y=0 TO 39 STEP 39
180 HLIN 0,39 AT Y
190 NEXT Y
200 FOR X=0 TO 39 STEP 39
210 VLIN 0,39 AT X
220 NEXT X
230 END
```

---

This program does the same graphics task as an earlier one written without the help of HLIN and VLIN statements. The program listing, itself, is not significantly shorter nor simpler, but it certainly executes the drawing operation faster. This speed is the advantage that HLIN and VLIN statements have over the PLOT statement.

**The SCRN Statement**     The SCRN statement fetches the color code of any low-resolution graphics point on the screen. The appropriate syntax is:

$$\text{SCRN}(x,y)$$

where x is the column number and y is the low-resolution row number.

The following example assigns the color code of a graphics element at location 12,35 to variable CC:

```
CC=SCRN(12,35)
```

**Graphics Techniques**     The small size of the family of low-resolution graphics statements belies its flexibility. Given the four statements, a programmer can:

- Compose elaborate, full-color static (nonanimated) pictures on the screen.
- Draw colorful and dynamic graphs.

- Plot combinations of static images and simple moving objects.
- Compose interesting and entertaining full-color animation sequences.

Composing static pictures is a matter of planning the size, shape, position, and color of each of the elements of the picture, using a worksheet such as the one in Fig. 6-1 as a guide. After doing the planning on such a worksheet, the next step is to draw up sequences of COLOR, PLOT, HLLN, and VLLN statements for each element. The idea is simple in principle, but it requires some time and effort on your part. If you haven't seen many interesting low-resolution static pictures, it means that you haven't encountered anyone who cares to take the time to write such programs.

Making up programs for drawing graphs is also quite simple in principle. It is a matter of applying the numerical data you want to graph to the small family of low-resolution graphics statements. Colorful bar graphs are especially easy to work out on the Apple system.

Knowing how to create the impression of a simple moving object against a colorful background can lead to a lot of exciting graphics-oriented games. Moving that simple object on the screen is a matter of looking ahead to its next position, saving the color code of that point by means of a SCRN statement, plotting over the moving object with a previously saved color code, and then plotting the moving object in its new position.

Full-screen animation sequences are difficult to generate without noticing a lot of flickering on the screen. But the job can be done quite satisfactorily, especially if you can have access to the secondary low-resolution graphics page. Again, if you haven't seen any good full-screen animations on the Apple, it is because so few people have the necessary combination of artistic imagination, skill, and patience.

The exact details for implementing any of these graphics techniques are far beyond the scope of this book. The basic ideas are rather simple and Integer BASIC includes all the necessary tools; the problem is that illustrating the step-by-step procedures would fill an entire book.

**Role of the Apple Monitor** All of the low-resolution graphics statements in BASIC refer to the Apple monitor. The BASIC statements are simply keywords for accessing the actual graphics routines in the machine-language monitor. Dressing up the graphics techniques with BASIC keywords slows down the drawing operations. That will be no big surprise to you later on when we look at the graphics techniques again from a purely machine-language point of view.

**POKEING COLORS TO THE SCREEN** Just as it is possible, and often desirable, to POKE text characters to the video text memory, it is possible to POKE colors to the low-resolution memory.

**Organization of the Low-Resolution Video Memory** The low-resolution video memory is formatted in exactly the same way as the video text memory. In fact, they are one and the same. POKEing characters into the primary-page video memory under TEXT plots text characters. POKEing data into the primary-page video memory under GR plots colored rectangles of light.

Tables 6-2 and 6-3 show the memory maps for the primary and secondary low-resolution graphics RAM. The maps are shown here only as a matter of convenience; they are identical to the video text memory maps shown earlier in Tables 4-1 and 4-2.

**Table 6-2. Graphics Primary Page Memory Map**

Line	Address Range
0	1024–1063
1	1152–1191
2	1280–1319
3	1408–1447
4	1536–1575
5	1664–1703
6	1792–1831
7	1920–1959
8	1064–1103
9	1192–1231
10	1320–1359
11	1448–1487
12	1576–1615
13	1704–1743
14	1832–1871
15	1960–1999
16	1104–1143
17	1232–1271
18	1360–1399
19	1488–1527
20	1616–1655
21	1744–1783
22	1872–1911
23	2000–2039

**Table 6-3. Graphics Secondary Page Memory Map**

Line	Address Range
0	2048–2087
1	2176–2215
2	2304–2343
3	2432–2471
4	2560–2599
5	2688–2727
6	2816–2855
7	2944–2983
8	2088–2127
9	2216–2255
10	2344–2383
11	2472–2511
12	2600–2639
13	2728–2767
14	2856–2895
15	2984–3023
16	2128–2167
17	2256–2295
18	2384–2423
19	2512–2551
20	2640–2679
21	2768–2807
22	2896–2935
23	3024–3063

The following experiment ought to convince you that the video text and low-resolution graphics occupy the same RAM addresses and can be accessed in the same fashion.

1. Enter and run this text-oriented program:

---

```
10 TEXT
20 CALL -936
30 FOR N=0 TO 39
40 POKE 1448+N,153
50 NEXT N
60 GOTO 60
```

---

The program uses text techniques described in Chapter 4 to POKE a full line of 153 character codes (NORMAL-1 Y character) to video RAM addresses 1448 through 1487. It POKES a full line of 40 Y characters to the screen. Since line 60 loops to itself, you must do a CTRL-C to get out of the program.

2. Alter line 10 in that program to read GR instead of TEXT. Run the program, which now looks like this:

---

```
10 GR
20 CALL -936
30 FOR N=0 TO 39
40 POKE 1448+N,153
50 NEXT N
60 GOTO 60
```

---

You should see a wide orange bar stretching horizontally across the screen.

What is the difference between the program routines in Steps 1 and 2? Both steps POKE code 153 to successively higher video RAM addresses, from 1448 through 1487. Step 1, however, prints a series of white-on-black Y characters, while Step 2 transforms those same Y characters into orange rectangles of light.

3. Get out of the program in Step 3 by doing a CTRL-C.
4. Do a TEXT command from the keyboard. You will see that most of the upper section of the text screen is filled with inverse @ characters. But the row of Y characters is there, too.

Returning from GR to TEXT causes the Apple system to interpret that series of characters you POKEd into video RAM as text characters rather than graphic characters. What is the meaning of all those inverse @ characters? Recall that executing the GR statement clears the low-resolution screen to all black; the inverse @ character happens to be the text interpretation of graphic black.

The text interpretation of any graphics characters remains on the screen when going from GR to TEXT because the system does not automatically clear the screen when switching modes in that direction.

The differences between the TEXT and GR interpretations of character codes is the main subject of the next discussion. What is more important at this point is to realize that the video text and low-resolution graphics memories are one and the same. That includes the secondary text/graphics page as well.

So the overall effect of POKES and PEEKs to video memory depends on whether the system is running under the TEXT or the GR mode. POKEing characters to video memory under TEXT prints text characters to the screen. POKEing characters to video memory under GR plots low-resolution graphic blocks to the screen.

**Graphics Color Codes** Chapter 4 describes how it is possible to POKE a family of 256 different text character codes to the primary or secondary video RAM addresses. Those codes are numbered 0 through 255 and represent the entire range of printable characters, including inverse, flashing, NORMAL-1, and NORMAL-2 characters. (See Tables 4-3 through 4-5.)

Since the low-resolution graphics scheme uses the same general video environment, it follows that there are also 256 possible graphics codes that are numbered 0 through 255. POKEing those codes to the primary or secondary page plots different colors on the screen.

By way of an introduction to this idea, enter and run the following text-related program:

---

```
100 TEXT
110 CALL -936
120 TAB 1: VTAB 1
130 FOR CC=0 TO 15
140 GOSUB 200
150 NEXT CC
160 VTAB 21
170 END
200 VTAB CC+1
210 FOR N=0 TO 39
220 POKE 256* PEEK (41)+ PEEK (40)+N,CC*17
230 NEXT N
240 RETURN
```

---

The result appears to be largely meaningless. You should see rows of 16 different text characters: inverse @, inverse Q, inverse quotation mark, inverse 3, flashing D, and so on. They represent a series of text codes from 0 through 255 in steps of 17.

The routine uses TAB and VTAB statements to set the lines of text in an orderly fashion. Line 220 uses a procedure described earlier to POKE characters to the current RAM address calculated from BASH and BASL. It is the peculiar selection of character codes, and not the basic text-printing idea, that is new here.

Now, revise line 100 in that program to read GR instead of TEXT. The result should look like this:



---

```
100 GR
110 CALL -936
120 TAB 1: VTAB 1
130 FOR CC=0 TO 15
140 GOSUB 200
150 NEXT CC
160 VTAB 21
170 END
200 VTAB CC+1
210 FOR N=0 TO 39
220 POKE 256* PEEK (41)+ PEEK (40)+N,CC*17
230 NEXT N
240 RETURN
```

---

When you run this version of the same program, the results on the screen will be remarkably different and far more meaningful. You'll see the 16 low-resolution graphics colors plotted as horizontal bands. That "peculiar" selection of character codes plots lines of different colors under the GR mode of operation.

Table 6-4 summarizes those codes and the colors they create when POKEd to the screen in the GR mode and in the TEXT mode. Those code numbers, shown in increments of 17, represent the sequence of 16 colors that can be PLOTted to the screen. Compare them with the PLOT codes in Table 6-1.

Using that series of code numbers, you can plot full-sized blocks of color anywhere on the screen but at the four bottom lines. Notice that they are full-sized blocks, and not the half-height blocks that appear in response to a PLOT statement. Since you are getting into the low-resolution graphics mode by doing a GR, you cannot plot colors in the lower four lines that are dedicated to text operations. Try to POKE a color code in that text region, and you will see a text character.

POKEing colors to the low-resolution graphics screen is a bit trickier and more cumbersome than using the more traditional methods (using COLOR, PLOT, HLIN, and so on), but the notion offers some advantages as well. One advantage is that you can POKE color codes to the secondary video page. Like PRINT statements in the TEXT mode, PLOT-type statements in GR work only with the primary page of video. There are other instances when it is advantageous to combine traditional graphics statements and POKE statements.

The discussion to this point deals with only 16 different colors that can be POKEd to the graphics screen. There are supposed to be 256 possible color codes. What about the remaining 240 codes?

The 16 codes cited in Table 6-4 represent the codes that POKE a full block of a single color to the graphics screen. They are not the half-height

blocks that you can PLOT to the screen by the traditional graphics methods, but full-sized blocks that occupy an entire character space. What's more, the full-sized block has a single color. Therein lies the difference between the 16 codes in Table 6-4 and the remaining 240 of them.

Most codes that are POKEd to the graphics screen produce two-color blocks. They are a pair of half-sized blocks that have different colors. The following program lets you view all 256 color codes as they are POKEd in sequence to the graphics screen:

---

```
100 GR
110 CALL -936
120 TAB 1: VTAB 1
130 FOR N=0 TO 127
140 GOSUB 290
150 NEXT N
160 VTAB 21: TAB 1
170 PRINT "CODES 0-127": PRINT
180 INPUT "STRIKE RETURN FOR MORE ...",S$
190 CALL -936
200 VTAB 1: TAB 1
210 FOR N=128 TO 255
220 GOSUB 290
230 NEXT N
240 VTAB 21: TAB 1
250 PRINT "CODES 128-255": PRINT
260 INPUT "WANT TO SEE FIRST SERIES AGAIN (Y/N)?",S$
270 IF S$="N" THEN END
280 CALL -936: GOTO 120
290 POKE 256* PEEK (41)+ PEEK (40)+ PEEK (36),N
300 CALL -1036
310 CALL -1036
320 IF PEEK (36)=0 THEN CALL -926
330 RETURN
```

---

Enter and run the program, and you will see the first 128 color codes as they appear in the GR mode of operation. There will be a blank space between each code. You can view the other 128 codes by striking the RETURN key when prompted to do so. Notice that most of them are two-colored blocks.

Incidentally, you should be able to follow the techniques used in this particular program. They are really no different from the techniques used in TEXT programs. The only difference between this program and a program that prints out all of the Apple text characters, with a blank space between characters and a blank between rows, is the GR statement in line 100. Change that line to read TEXT, and you will see what we mean. Indeed, there are no real differences between the techniques for POKEing text (Chapter 4) and POKEing graphics.

**Table 6-4. Full Block Color Codes**

<b>POKE color code</b>	<b>Full-block color</b>	<b>Equivalent text character</b>
0	black	inverse @
17	magenta	inverse Q
34	dark blue	inverse "
51	purple	inverse 3
68	dark green	flashing D
85	grey 1	flashing U
102	medium blue	flashing &
119	light blue	flashing 7
136	brown	NORMAL-1 H
153	orange	NORMAL-1 Y
170	grey 2	NORMAL-1 *
187	pink	NORMAL-1 ;
204	light green	NORMAL-2 L
221	yellow	NORMAL-2 ]
238	aquamarine	NORMAL-2 .
255	white	NORMAL-2 ?

Table 6-5 lists all possible combinations of upper and lower colors and the corresponding color codes for all 256 POKE values. It is an extensive list, to be sure, but it is indispensable for planning low-resolution color graphics. To determine which codes to use from the table, first decide on the colors you want for the upper and lower halves of the block. Search the UPPER part of the listing first for the upper color. After finding the proper upper color codes, search that section for the color code of the LOWER half of the block.

Suppose, for instance, you want to plot a purple segment over an orange segment at video address 1068. First search the table for the PURPLE listing as the UPPER portion of the block. After you find that, look for the ORANGE listing as the LOWER portion in that section of the table. The corresponding color code is 147. Since that purple-over-orange graphic is to appear at address 1068, the appropriate statement for doing the job is:

**POKE 1068,147**

Try it. Get into the GR mode and execute the statement.

**Table 6-5. Upper/Lower Color Codes**

UPPER/LOWER COLOR	CODE
BLACK/BLACK	0
BLACK/MAGENTA	16
BLACK/DARK BLUE	32
BLACK/PURPLE	48
BLACK/DARK GREEN	64
BLACK/GREY 1	80
BLACK/MEDIUM BLUE	96
BLACK/LIGHT BLUE	112
BLACK/BROWN	128
BLACK/ORANGE	144
BLACK/GREY 2	160
BLACK/PINK	176
BLACK/LIGHT GREEN	192
BLACK/YELLOW	208
BLACK/AQUA	224
BLACK/WHITE	240
MAGENTA/BLACK	1
MAGENTA/MAGENTA	17
MAGENTA/DARK BLUE	33
MAGENTA/PURPLE	49
MAGENTA/DARK GREEN	65
MAGENTA/GREY 1	81
MAGENTA/MEDIUM BLUE	97
MAGENTA/LIGHT BLUE	113
MAGENTA/BROWN	129
MAGENTA/ORANGE	145
MAGENTA/GREY 2	161
MAGENTA/PINK	177
MAGENTA/LIGHT GREEN	193
MAGENTA/YELLOW	209
MAGENTA/AQUA	225
MAGENTA/WHITE	241
DARK BLUE/BLACK	2
DARK BLUE/MAGENTA	18
DARK BLUE/DARK BLUE	34
DARK BLUE/PURPLE	50
DARK BLUE/DARK GREEN	66
DARK BLUE/GREY 1	82
DARK BLUE/MEDIUM BLUE	98

**Table 6-5—cont. Upper/Lower Color Codes**

<b>UPPER/LOWER COLOR</b>	<b>CODE</b>
DARK BLUE/LIGHT BLUE	114
DARK BLUE/BROWN	130
DARK BLUE/ORANGE	146
DARK BLUE/GREY 2	162
DARK BLUE/PINK	178
DARK BLUE/LIGHT GREEN	194
DARK BLUE/YELLOW	210
DARK BLUE/AQUA	226
DARK BLUE/WHITE	242
PURPLE/BLACK	3
PURPLE/MAGENTA	19
PURPLE/DARK BLUE	35
PURPLE/PURPLE	51
PURPLE/DARK GREEN	67
PURPLE/GREY 1	83
PURPLE/MEDIUM BLUE	99
PURPLE/LIGHT BLUE	115
PURPLE/BROWN	131
PURPLE/ORANGE	147
PURPLE/GREY 2	163
PURPLE/PINK	179
PURPLE/LIGHT GREEN	195
PURPLE/YELLOW	211
PURPLE/AQUA	227
PURPLE/WHITE	243
DARK GREEN/BLACK	4
DARK GREEN/MAGENTA	20
DARK GREEN/DARK BLUE	36
DARK GREEN/PURPLE	52
DARK GREEN/DARK GREEN	68
DARK GREEN/GREY 1	84
DARK GREEN/MEDIUM BLUE	100
DARK GREEN/LIGHT BLUE	116
DARK GREEN/BROWN	132
DARK GREEN/ORANGE	148
DARK GREEN/GREY 2	164
DARK GREEN/PINK	180
DARK GREEN/LIGHT GREEN	196
DARK GREEN/YELLOW	212

**Table 6-5—cont. Upper/Lower Color Codes**

UPPER/LOWER COLOR	CODE
DARK GREEN/AQUA	228
DARK GREEN/WHITE	244
GREY 1/BLACK	5
GREY 1/MAGENTA	21
GREY 1/DARK BLUE	37
GREY 1/PURPLE	53
GREY 1/DARK GREEN	69
GREY 1/GREY 1	85
GREY 1/MEDIUM BLUE	101
GREY 1/LIGHT BLUE	117
GREY 1/BROWN	133
GREY 1/ORANGE	149
GREY 1/GREY 2	165
GREY 1/PINK	181
GREY 1/LIGHT GREEN	197
GREY 1/YELLOW	213
GREY 1/AQUA	229
GREY 1/WHITE	245
MEDIUM BLUE/BLACK	6
MEDIUM BLUE/MAGENTA	22
MEDIUM BLUE/DARK BLUE	38
MEDIUM BLUE/PURPLE	54
MEDIUM BLUE/DARK GREEN	70
MEDIUM BLUE/GREY 1	86
MEDIUM BLUE/MEDIUM BLUE	102
MEDIUM BLUE/LIGHT BLUE	118
MEDIUM BLUE/BROWN	134
MEDIUM BLUE/ORANGE	150
MEDIUM BLUE/GREY 2	166
MEDIUM BLUE/PINK	182
MEDIUM BLUE/LIGHT GREEN	198
MEDIUM BLUE/YELLOW	214
MEDIUM BLUE/AQUA	230
MEDIUM BLUE/WHITE	246
LIGHT BLUE/BLACK	7
LIGHT BLUE/MAGENTA	23
LIGHT BLUE/DARK BLUE	39
LIGHT BLUE/PURPLE	55

**Table 6-5—cont. Upper/Lower Color Codes**

UPPER/LOWER COLOR	CODE
LIGHT BLUE/DARK GREEN	71
LIGHT BLUE/GREY 1	87
LIGHT BLUE/MEDIUM BLUE	103
LIGHT BLUE/LIGHT BLUE	119
LIGHT BLUE/BROWN	135
LIGHT BLUE/ORANGE	151
LIGHT BLUE/GREY 2	167
LIGHT BLUE/PINK	183
LIGHT BLUE/LIGHT GREEN	199
LIGHT BLUE/YELLOW	215
LIGHT BLUE/AQUA	231
LIGHT BLUE/WHITE	247
BROWN/BLACK	8
BROWN/MAGENTA	24
BROWN/DARK BLUE	40
BROWN/PURPLE	56
BROWN/DARK GREEN	72
BROWN/GREY 1	88
BROWN/MEDIUM BLUE	104
BROWN/LIGHT BLUE	120
BROWN/BROWN	136
BROWN/ORANGE	152
BROWN/GREY 2	168
BROWN/PINK	184
BROWN/LIGHT GREEN	200
BROWN/YELLOW	216
BROWN/AQUA	232
BROWN/WHITE	248
ORANGE/BLACK	9
ORANGE/MAGENTA	25
ORANGE/DARK BLUE	41
ORANGE/PURPLE	57
ORANGE/DARK GREEN	73
ORANGE/GREY 1	89
ORANGE/MEDIUM BLUE	105
ORANGE/LIGHT BLUE	121
ORANGE/BROWN	137
ORANGE/ORANGE	153
ORANGE/GREY 2	169

**Table 6-5—cont. Upper/Lower Color Codes**

UPPER/LOWER COLOR	CODE
ORANGE/PINK	185
ORANGE/LIGHT GREEN	201
ORANGE/YELLOW	217
ORANGE/AQUA	233
ORANGE/WHITE	249
GREY 2/BLACK	10
GREY 2/MAGENTA	26
GREY 2/DARK BLUE	42
GREY 2/PURPLE	58
GREY 2/DARK GREEN	74
GREY 2/GREY 1	90
GREY 2/MEDIUM BLUE	106
GREY 2/LIGHT BLUE	122
GREY 2/BROWN	138
GREY 2/ORANGE	154
GREY 2/GREY 2	170
GREY 2/PINK	186
GREY 2/LIGHT GREEN	202
GREY 2/YELLOW	218
GREY 2/AQUA	234
GREY 2/WHITE	250
PINK/BLACK	11
PINK/MAGENTA	27
PINK/DARK BLUE	43
PINK/PURPLE	59
PINK/DARK GREEN	75
PINK/GREY 1	91
PINK/MEDIUM BLUE	107
PINK/LIGHT BLUE	123
PINK/BROWN	139
PINK/ORANGE	155
PINK/GREY 2	171
PINK/PINK	187
PINK/LIGHT GREEN	203
PINK/YELLOW	219
PINK/AQUA	235
PINK/WHITE	251
LIGHT GREEN/BLACK	12
LIGHT GREEN/MAGENTA	28



**Table 6-5—cont. Upper/Lower Color Codes**

<b>UPPER/LOWER COLOR</b>	<b>CODE</b>
LIGHT GREEN/DARK BLUE	44
LIGHT GREEN/PURPLE	60
LIGHT GREEN/DARK GREEN	76
LIGHT GREEN/GREY 1	92
LIGHT GREEN/MEDIUM BLUE	108
LIGHT GREEN/LIGHT BLUE	124
LIGHT GREEN/BROWN	140
LIGHT GREEN/ORANGE	156
LIGHT GREEN/GREY 2	172
LIGHT GREEN/PINK	188
LIGHT GREEN/LIGHT GREEN	204
LIGHT GREEN/YELLOW	220
LIGHT GREEN/AQUA	236
LIGHT GREEN/WHITE	252
 YELLOW/BLACK	 13
YELLOW/MAGENTA	29
YELLOW/DARK BLUE	45
YELLOW/PURPLE	61
YELLOW/DARK GREEN	77
YELLOW/GREY 1	93
YELLOW/MEDIUM BLUE	109
YELLOW/LIGHT BLUE	125
YELLOW/BROWN	141
YELLOW/ORANGE	157
YELLOW/GREY 2	173
YELLOW/PINK	189
YELLOW/LIGHT GREEN	205
YELLOW/YELLOW	221
YELLOW/AQUA	237
YELLOW/WHITE	253
 AQUA/BLACK	 14
AQUA/MAGENTA	30
AQUA/DARK BLUE	46
AQUA/PURPLE	62
AQUA/DARK GREEN	78
AQUA/GREY 1	94
AQUA/MEDIUM BLUE	110
AQUA/LIGHT BLUE	126
AQUA/BROWN	142

**Table 6-5—cont. Upper/Lower Color Codes**

UPPER/LOWER COLOR	CODE
AQUA/ORANGE	158
AQUA/GREY 2	174
AQUA/PINK	190
AQUA/LIGHT GREEN	206
AQUA/YELLOW	222
AQUA/AQUA	238
AQUA/WHITE	254
WHITE/BLACK	15
WHITE/MAGENTA	31
WHITE/DARK BLUE	47
WHITE/PURPLE	63
WHITE/DARK GREEN	79
WHITE/GREY 1	95
WHITE/MEDIUM BLUE	111
WHITE/LIGHT BLUE	127
WHITE/BROWN	143
WHITE/ORANGE	159
WHITE/GREY 2	175
WHITE/PINK	191
WHITE/LIGHT GREEN	207
WHITE/YELLOW	223
WHITE/AQUA	239
WHITE/WHITE	255

**Getting Help From the Cursor Registers** Of course, you can plan an elaborate, full-color picture to be drawn in low-resolution graphics, use the table to determine the necessary color codes, and then write a program that POKEs those codes to the appropriate video RAM addresses. Even for ambitious graphics programmers, that represents a lot of tedious work. It would be nice to simplify matters somewhere along the line. Fortunately, there is at least one technique that can simplify one major phase of the task.

There is no need to POKE the graphics codes to absolute video memory addresses; at least there is no need for figuring out a RAM address for each and every graphic code to be POKEd to the screen. Since the text and graphics modes share the same video memory, it is altogether possible, and quite practical, to use many of the cursor-related operations already described for purely text-oriented programs. You can see some of the ideas in some previous demonstration programs.

The key to applying cursor-type functions to low-resolution graphics is the function:

## 256\*PEEK(41)+PEEK(40)+PEEK(36)

This function generates the actual video RAM address based upon the content of BASH, BASL and CH. It turns up a valid POKE address for a text character or graphic color code.

The function can then be used in conjunction with several cursor-moving CALLs:

CALL -926 — Linefeed/carriage return.  
CALL -1036 — Advance.  
CALL -1008 — Backspace  
CALL -922 — Downward linefeed.  
CALL -998 — Upward linefeed.

And the function can be used with TAB and VTAB statements as well. In fact, the only cursor-related functions that do not work well with low-resolution graphics are those that clear a portion or all of the screen. (A CALL -868 in the TEXT mode erases to the end of the current line. Executed in GR, it leaves black-over-grey bars to the end of the line.)

The following program plots a large orange square near the middle of the screen. It uses TAB and VTAB to establish the position of the upper left-hand corner of the square, the ADVANCE function to plot the orange codes along each line, and a CALL -922 to do a downward linefeed at the beginning of each line.

---

```
100 GR
110 CALL -936
120 VTAB 8: TAB 18
130 FOR Y=0 TO 5
140 FOR X=0 TO 7
150 POKE 256* PEEK (41)+ PEEK (40)+ PEEK (36),153
160 CALL -1036
170 NEXT X
180 TAB 18
190 CALL -922
200 NEXT Y
210 VTAB 20
220 END
```

---

The scheme works nicely as long as the cursor remains within the graphics area of the screen. However, if you allow it to move out of the graphics area and into the four-line text area at the bottom of the screen, the system prints the text version of the characters rather than the graphic version of them.

So what is the purpose of the VTAB 20 statement in program line 210? Its purpose is to force the cursor out of the graphics area so that the subsequent END statement prints the prompt symbol in the text area. Delete line 210 from the program and run it again. The little pattern of colors added to the display represents the GR version of the prompt symbol.

**ALTERNATIVE SCREEN FORMATS** Executing the GR command sets up the normal low-resolution graphics format. It displays the primary text/graphics page, allocates most of the screen for graphics operations, and leaves four lines for text at the bottom of the screen. There are some desirable alternatives, however.

For one, you might want to work with a full screen of graphics, getting rid of that lower four lines of text that can be an aesthetic nuisance at times. Or perhaps you want to switch from the TEXT to low-resolution graphics without having the system automatically clear the graphics portion of the screen. You may also want that page of low-resolution graphics.

A few simple POKE statements will give you that control over the video text/graphics system.

**Table 6-6. Text/Graphics Software Switches**

POKE Address	Video Mode
-16299	Display the secondary page
-16300	Display the primary page
-16301	Display mixed text and graphics
-16302	Display all text or all graphics
-16303	Display a text mode
-16304	Display a graphics mode

**The Screen Mode "Switches"** Table 6-6 shows three pairs of text/graphics "software switches." Together, they make up a selection of three video modes. You can set one of the two conditions in each mode by POKEing a 0 to the designated address.

The first pair of software switches uses addresses -16299 and -16300. They determine whether the system displays the secondary or primary page of video information. It makes no difference whether you are working

POKE -16299,0—Display the *secondary* video page.

POKE -16300,0—Display the *primary* video page.

in the text or graphics modes. POKEing to these addresses sets up the display for the primary or secondary page.

POKEing a 0 to either of those addresses sets the corresponding page to the video screen and automatically resets its counterpart.

The second pair of addresses, -16301 and -16302, works together this way:

POKE -16301,0—Display mixed text and graphics.

The bottom four lines are dedicated to text and the remainder of the screen is set for low-resolution graphics.

POKE -16302,0—Display all text or all graphics.

Doing a GR command sets up the mixed text/graphics mode and automatically clears the graphics portion of the screen to black. A POKE -16301,0 also sets up the mixed text/graphics mode, but it *does not* clear the graphics part of the screen.

The third pair of software “switches” lets you determine whether you are working in a text or graphics mode.

Having three pairs of software mode switches available means that there are eight possible combinations of mode settings. Once you set up one of those combinations, the system remains in that mode until you do something to change it. A programmer rarely has to think in terms of eight separate video mode combinations. Rather, a programmer thinks in terms of POKE operations that are necessary for going from one mode combination to another. You will see this notion at work in the following discussions.

This program lets you play around with all the various video modes. Enter the program and RUN it. It is a key-controlled program that responds immediately to the following keystrokes:

S —Displays the secondary page.

P —Displays the primary page.

M —Displays mixed text/graphics.

F —Displays all (full-screen) text/graphics.

T —Sets a text mode.

G —Sets a graphics mode.

The program makes it possible for you to set up all possible combinations of pages and text/graphics modes. Study the program carefully, and you will see how it takes advantage of the information in Table 6-6.

---

```
100 K= PEEK (-16384)
110 IF K<128 THEN 100
120 POKE -16368,0
130 IF K=208 THEN 230
140 IF K=211 THEN 240
150 IF K=212 THEN 250
160 IF K=199 THEN 210
170 IF K=198 THEN 200
180 IF K=205 THEN 220
190 GOTO 100
200 POKE -16302,0: GOTO 100
210 POKE -16304,0: GOTO 100
220 POKE -16301,0: GOTO 100
230 POKE -16300,0: GOTO 100
240 POKE -16299,0: GOTO 100
250 POKE -16303,0: GOTO 100
```

---

**Full-Screen Graphics** Suppose that the system is in the normal, full-screen text mode and is displaying the primary video page. If you execute a GR statement from that mode, the system goes to mixed text/graphics and clears the graphics portion of the primary page. But suppose that you want to go from the normal, full-screen text mode to a full-screen, low-resolution graphics mode. You want to work with low-resolution graphics, but without having the four lower lines on the screen allocated for text-only operations. Here is a combination of POKES that accomplish that feat:

```
POKE -16302,0
POKE -16304,0
```

The horizontal dimensions of the graphics area are identical for mixed and full-screen graphics. The range of vertical plotting is greater for full-screen modes, however. Instead of being limited to vertical PLOT coordinates from 0 through 39, with full-screen graphics you can use coordinates from 0 through 47. Now, the four lower lines on the screen are open to low-resolution operations. In the same fashion, full-screen graphics extends the vertical range of VLIN statements to 0 through 47.

Try the following full-screen graphics program:

---

```
100 POKE -16302,0
110 POKE -16304,0
120 COLOR=4
130 FOR Y=0 TO 47
140 HLIN 0,39 AT Y
150 NEXT Y
160 COLOR=15
170 FOR Y=0 TO 47 STEP 47
180 HLIN 0,39 AT Y
190 NEXT Y
200 FOR X=0 TO 39 STEP 39
210 VLIN 0,47 AT X
220 NEXT X
230 GOTO 230
```

---

The program plots a green background and surrounds it with a white border. Notice that the image fills the entire screen. HLIN operations are still limited to coordinates from 0 through 39, but the technique extends the VLIN operations to coordinates from 0 through 47.

When using full-screen, low-resolution graphics, it is especially important that the program doesn't come to an end before you want it to. Line 230 in the preceding program, for instance, does a "loop-to-self" to prevent the program from executing an END-type operation. Delete line 230 or replace it with an END statement, run the program again, and notice the undesirable result.

Try appending the program with these lines:

---

```
230 FOR T=0 TO 1000: NEXT T
240 TEXT
250 CALL -936
260 END
```

---

When you run the program modified in that fashion, the image is displayed for a period determined by the delay loop in line 230. After that, line 240 returns the system to the text mode, line 250 clears the screen and homes the cursor, and line 260 brings the program to an end.

Line 240 in that modified version of the program demonstrates that you can always return to a normal TEXT mode from full-screen graphics by executing the TEXT statement. Alternatively, you can accomplish the same thing by doing a POKE -16303,0. Why is this so? Replace line 240 with POKE -16303,0 and note the effect on the operation of the program.

To recapitulate, there are two ways to return from full-screen graphics to full-screen text: TEXT or POKE -16303,0.

Just as the elementary low-resolution graphics statements work in full-screen modes, so do the POKE-to-graphics techniques work. It is possible to POKE color codes into those four lower lines on the screen and see the color blocks appear there. There is one note of caution when POKEing color codes to full-screen graphics: if you are using the 256\*PEEK(41)+PEEK(40)+PEEK(36) method, *avoid the last character space in the last line on the screen*. POKEing a color code into that particular position will cause the entire graphics screen to scroll upward.

Going to full-screen graphics from the normal text mode does not automatically clear the screen to black. This poses no real problem if your drawing routine refers to the entire screen, but there are occasions when you will want to clear the graphics screen to black before beginning a drawing routine. The following program suggests a method for switching to full-screen graphics and clearing to black:

---

```
100 POKE -16302,0: POKE -16304,0
110 COLOR=0
120 FOR X=0 TO 39
130 VLIN 0,47 AT X
140 NEXT X
150 GOTO 150
```

---

See if you can figure out how and why it works. Try it for yourself.

**Working With the Secondary Page** Doing a POKE -16299,0 brings the secondary video page to the screen; and according to Table 6-6, doing a POKE -16300,0 brings back the primary page. Whether you see a text or low-resolution graphics mode on those pages depends on whether you have POKEd a 0 to address -16303 (for text) or -16304 (for graphics). What's more, you can choose full-screen text or graphics by doing a POKE -16302,0 or mixed text/graphics by doing a POKE -16301,0. Obviously there are a lot of options here.

Before demonstrating some of these options, it is important to recall some of the main features of the secondary video page. Chapter 4 outlined those features for text operations. The same features apply to low-resolution graphics as well.



First, you must do a LOMEM:3072 if you plan to work with the secondary video page from BASIC. The execution of Integer BASIC often places variables into RAM addresses used for the secondary page of video. Doing the LOMEM:3072 forces BASIC to use RAM space above the secondary page memory.

Second, it is not possible to PRINT text directly to the secondary page. The best technique offered thus far for plotting text characters to the secondary page is to POKE their codes into that video RAM area. Likewise, you cannot directly use PLOT, HLIN, VLIN, or SCRNB on the secondary page. At this point in the book, the best way to draw graphics to the secondary page is with POKE-color techniques.

Finally, it is important to know that the secondary page for low-resolution graphics occupies the same RAM addresses as the secondary page for text. (See Table 6-3.) The video RAM addresses for the secondary page are equal to those of the primary page plus 1024.

After executing a LOMEM:3072, enter and run the following program:

---

```
10 REM  ** THIS PROGRAM LATCHES UP IF YOU FAIL TO DO A LO
MEM:3071 FIRST **
15 POKE -16302,0: POKE -16304,0
20 VTAB 1: TAB 1
30 FOR N=0 TO 959
40 CP=256* PEEK (41)+ PEEK (40)+ PEEK (36)
45 CALL -1036
50 POKE CP,0: POKE CP+1024,0
60 NEXT N
70 END
```

---

The program clears both the primary and secondary graphics pages to black. You will most likely be watching the primary page during the execution of this rather slow-running program, but you can check out the effect on the secondary page by doing a POKE -16299,0 from the keyboard after the program ends. Line 50 is responsible for clearing both pages to black.

The routine illustrates two points. First, you can POKE graphics to the secondary page by thinking in terms of primary-page graphics and adding 1024 to the POKE addresses. Second, the drawing procedure is terribly slow. Even if you shortened line 50 to POKE to just one of the pages, the program wouldn't run much faster. The slow drawing speed of POKE graphics can be made more tolerable by displaying a finished drawing on one page while the program is drawing the text image on the other page.

Another useful trick is to use the primary graphics page for the faster drawing operations of PLOT, HLIN, and VLIN, and the secondary page for the slower POKE graphics.

The situation doesn't have to be so complicated, however. You will find in a later chapter that machine-language graphics run at the same high speed on both the primary and secondary pages. Animated graphics, in fact, demand the higher-speed, machine-language procedures.

# The High-Resolution Graphics Environment

The high-resolution graphics environment is fertile ground for a lot of intriguing experiments and programming work. It seems to be the least-used scheme for most people, though, because it is the least developed part of the Apple in terms of programmer convenience. **7**

If you have had some difficulty in the past working with high-resolution graphics programs, you aren't alone. Most beginners have trouble with it. However, the Programmer's Aid hi-res routines (included in the Integer BASIC ROMs) help a lot. At least they allow you to approach high-resolution graphics from BASIC. But, even then, hi-res programming operations seem quite peculiar and often confusing to BASIC programmers.

DOS users have an additional problem in that DOS boots up in sections of memory that are otherwise used for hi-res graphics. In other words, DOS and hi-res compete for RAM space.

This does not mean that high-resolution graphics is a no-man's land. It does mean that hi-res requires special programming care and precise thinking every step along the way.

**RECKONING WITH LOMEM AND HIMEM** When working with high-resolution graphics from Integer BASIC, you need to reckon with LOMEM and HIMEM. These two commands set the low memory and high memory locations, respectively, of BASIC programs and related data.

Exactly how you should handle the LOMEM and HIMEM settings depends a great deal on how much RAM is installed in your system. A brief discourse on the nature of LOMEM and HIMEM operations will help you understand how they affect your RAM space and high-resolution graphics operations.

**HIMEM Settings** Unless you direct the system to do otherwise, it will automatically set HIMEM to the highest available RAM address plus 1. That is called the *default* HIMEM address. In other words, if you have 16K of RAM, the default HIMEM setting is 16384. If you have 32K of RAM, the default HIMEM setting is 32768. Finally, if you have 48K of RAM, the default HIMEM setting is -16384. The system sets up its default HIMEM address whenever you initialize Integer BASIC with a CTRL-B operation.

Integer BASIC programs *always* begin from the HIMEM address, minus 1, and *build downward* toward lower addresses in RAM. So unless you make a special effort to set HIMEM to something other than its default value, Integer BASIC programs will always begin at the highest available RAM address and build downward.

If you have a 16K system, your highest available RAM address is right at the top of the primary page of high-resolution graphics. That's terrible! Not only do you lack a secondary page for high-resolution work, but Integer BASIC programming writes directly into the primary page that you do have. Therefore, you cannot hope to use hi-res and Integer BASIC together in a 16K system as long as HIMEM is at its default setting.

There is a way around the problem, though, and that is by entering a HIMEM:8192 prior to writing Integer BASIC programs for high-resolution graphics. Entering that command sets HIMEM to the bottom of the high-resolution video RAM area. Integer BASIC programs will then build downward and away from that vital graphics area. However, whenever you restart the Apple or do a CTRL-B, you must enter HIMEM:8192 again.

The HIMEM setting is a bit less critical if you have a 32K or 48K system. In those instances, the default HIMEM settings are well above even the secondary page of hi-res video RAM. With a 32K system, the default HIMEM setting leaves 1K of useful RAM and an additional 1K if you decide you won't need the secondary page of hi-res graphics. (If a total of 2K of Integer BASIC programming RAM doesn't seem to be enough, that's too bad. You'd better think about buying some more.) The default HIMEM setting for a 48K system places the BASIC programming 24K above any video memory, so that there is rarely any need to worry about running out of RAM. Perhaps Apple engineers had a 48K system in mind when they worked out the high-resolution graphics schemes.

**LOMEM Settings** The default setting for the LOMEM address is always 2048, regardless of the amount of RAM in the Apple system. This address marks the starting point of the variable table, which is used for storing the values of variables that are generated during the execution of an Integer BASIC program. The variable table builds *upward* from LOMEM. So as you enter an Integer BASIC program, the program instructions begin

at HIMEM and build downward. As you execute that program, any variables that it uses begin at LOMEM and build upward. The unused RAM in between is narrowed down from both ends at the same time.

The default LOMEM setting is well below any high-resolution graphics RAM space, so it would take a very unusual program to cause the Integer BASIC tables to encroach on high-resolution graphics memory.

Another important feature of LOMEM is that it specifies the default entry address of shape tables. You can force the shape tables to be loaded elsewhere, as explained later; otherwise they will load from cassette tape to the LOMEM address and upward.

Now think about this: If Integer BASIC stores its variables from LOMEM and up, and if shape tables load from cassette tape from LOMEM and up, won't there be a conflict of RAM space? Not if you load the shape tables from tape as spelled out in the Apple manuals.

Indeed, shape tables begin loading at the LOMEM address (unless you clearly specify otherwise). But as the tables are loaded, the system pushes the LOMEM address above the shape tables. So after loading a shape table, LOMEM will have some higher address value. Then, BASIC's variable table begins from the new LOMEM. That represents a nice piece of software engineering.

One small matter: The default LOMEM address of 2048 puts it right at the beginning of the secondary text/low-resolution graphics page. You are in trouble if you plan to use that secondary page of low-resolution graphics. But again, there is a way around that problem, and that is to set LOMEM at the top of the secondary low-res video RAM by entering a LOMEM:3072.

Suppose that you do set LOMEM above the low-res video memory. What happens when you load a hi-res shape table? No problem. The table begins loading at your LOMEM address and pushes LOMEM upward from there.

**Some Recommended LOMEM and HIMEM Settings** Tables 7-1 and 7-2 show six arrangements of graphics modes. The first arrangement in each table is the simplest. It consists of only the primary page of low-resolution graphics. The last arrangement is the worst. It consists of both the primary and secondary pages of both high- and low-resolution graphics.

The columns labeled LOMEM Setting and HIMEM Setting recommend what you should do, if anything, prior to entering the Integer BASIC programming. The final column, Usable RAM, shows how much RAM is then available for Integer BASIC programs and hi-res shape tables.

Table 7-1 applies to 16K systems, while Table 7-2 applies to 32K and 48K systems.

**Table 7-1. 16K LOMEM and HIMEM Settings**

<b>Graphics Mode Combination</b>	<b>LOMEM Setting</b>	<b>HIMEM Setting</b>	<b>Usable RAM</b>
pri low-res	default <sup>1</sup>	default <sup>2</sup>	14K
pri low-res sec low-res	LOMEM:3072	default	13K
pri low-res pri hi-res	default	HIMEM:8192	6K
pri low-res sec low-res pri hi-res	LOMEM:3072	HIMEM:8192	5K
pri low-res pri hi-res sec hi-res	cannot be done		
pri low-res sec low-res pri hi-res sec hi-res	cannot be done		

<sup>1</sup>Default LOMEM is 2048

<sup>2</sup>Default HIMEM is 16384

Suppose that you are using a 16K Apple system and you want to do some graphics on both the primary and secondary pages of low resolution, and a bit of primary-page hi-res work as well. According to Table 7-1, you should begin by doing LOMEM:3072 and HIMEM:8192. Doing that from the keyboard (in BASIC's command mode) leaves 5K of RAM for BASIC programming and hi-res shape tables.

Or if you have a 48K system and want to use both pages of low- and high-resolution graphics, you should set up the memory system by doing a LOMEM:3072. There is no need to change the default HIMEM address in this particular instance. That leaves about 45K of RAM for programming and shape tables. (Doing the same thing on a 32K system leaves 29K of usable RAM.)

**Programming LOMEM and HIMEM Addresses** Integer BASIC does not support LOMEM and HIMEM statements within a program listing. That is unfortunate, because it would be nice to set LOMEM and HIMEM at the beginning of a program, eliminating the need for having to set it before loading the program.

Actually, it is possible to set LOMEM at the beginning of an Integer BASIC program. The technique takes advantage of the fact that the

**Table 7-2. 32K and 48K LOMEM and HIMEM Settings**

<b>Graphics Mode Combination</b>	<b>LOMEM Setting</b>	<b>HIMEM Setting</b>	<b>Usable RAM</b>
pri low-res	default <sup>1, 3</sup>	default <sup>2, 4</sup>	30K/46K
pri low-res sec low-res	LOMEM:3072	default	29K/45K
pri low-res pri hi-res	default	default	30K/46K
pri low-res sec low-res pri hi-res	LOMEM:3072	default	29K/45K
pri low-res pri hi-res sec hi-res	default	default	30K/46K
pri low-res sec low-res pri hi-res sec hi-res	LOMEM:3072	default	29K/45K

<sup>1</sup>32K Default LOMEM is 2048

<sup>3</sup>48K Default LOMEM is 2048

<sup>2</sup>32K Default HIMEM is 32768

<sup>4</sup>48K Default HIMEM is -16384

LOMEM is carried in two successive memory addresses, 74 and 75. Those two RAM locations, known as LOMEML and LOMEMH, represent a 2-byte LOMEM address. Try this experiment:

1. Do a CTRL-B to initialize Integer BASIC.
2. Enter:

**PRINT PEEK(74),PEEK(75)**

You should see the result printed as

**0 8**

That is the 2-byte rendition of the current LOMEM address, and that translates into ordinary decimal numeration as address 2048. (See Appendix A if you are not sure about how to convert 2-byte decimal numbers into ordinary decimal format.)

Indeed, initializing Integer BASIC sets LOMEM to address 2048 as described earlier in this chapter.

3. Enter:

POKE 74,0 : POKE 75,12

POKEing those numbers into LOMEML and LOMEMH sets LOMEM to 3072—one of the LOMEM addresses recommended for many graphics applications.

Thus, you have the option of either setting LOMEM from the keyboard prior to loading a program—by using the LOMEM:3072 command—or you can write the POKE statements just cited as the *first* line in a BASIC program that uses that LOMEM setting. If you specify any ordinary BASIC variables before POKEing in the LOMEM setting, those variables will be lost to the program.

The current LOMEM setting is carried in RAM addresses 74 and 75. The first, LOMEML, is the low-order byte and the second, LOMEMH, is the high-order byte.

Unfortunately, the HIMEM address still has to be set from the keyboard prior to loading a program. Attempting to set HIMEM in any fashion after you've started loading some Integer BASIC programming will confuse the system. Most, if not all, of the prior BASIC programming will be lost.

There is a positive side to the picture, however. Not many programming situations call for setting HIMEM anywhere but at its default address, and that means you need not change it at all.

But for the sake of completeness, you ought to know that HIMEM is carried as a 2-byte number in memory locations 76 and 77.

The current HIMEM setting is carried in RAM addresses 76 and 77. The first, HIMEML, is the low-order byte and the second, HIMEMH, is the high-order byte.

Although it is possible to adjust the HIMEM setting or PEEK at its current value, it must be set prior to loading any Integer BASIC programming that uses it. Unless you are willing to lose existing BASIC programming, you should not adjust HIMEM after that programming is loaded into the system.



**INITIALIZING THE HI-RES SYSTEM** After dealing with the LOMEM and HIMEM settings as required for the sort of graphics job you want to do, the next step is to write a series of at least two initialization steps. The first defines some critical hi-res variables, and the second calls a Programmer's Aid hi-res initialization routine.

**Defining the Hi-Res Variables** The first step in the initialization procedure is to define some critical hi-res variables. This must be done *before* writing any BASIC statements that include other variable names. BASIC normally allows a programmer to define variables as they are needed, but this initialization scheme demands defining certain hi-res variables ahead of time.

There are as many as six different variables that must be defined at the beginning of a hi-res BASIC program. You can name them just about anything you choose, within certain limits. The variables are summarized in Table 7-3.

You are free to select alternative names for those variables, but your *names must have exactly the same number of characters as the original name*. You can, for example, use HX in place of XX; or you can use XO or NN. But you cannot use a variable name that has just one character or more than two. For instance, neither X nor XPLACE will work.

The same rule applies to any custom names for the COLR variable name. It must be a four character variable name. And if you don't happen to like the variable name of SHAPE, you can replace it with any other five character name.

A second important rule is that *the variables must be defined in the same sequence as shown in Table 7-3*. Defining them in any other order will confuse the hi-res system.

**Table 7-3. Hi-Res Variables**

Variable Name	Definition
XX	Horizontal component of a point's position on the hi-res screen (0-279).
YY	Vertical component of a point's position on the hi-res screen (0-159).
COLR	Color code for a point or line to be plotted on the hi-res screen (see Table 7-4).
SHAPE	Numerical value assigned to one of 255 possible shapes that are stored in shape tables.
ROT	Rotation factor for a figure that is drawn from a shape table.
SCALE	Scale factor for a figure that is drawn from a shape table.

You need not define all of those variables, however. If you do not plan to use shape tables in your hi-res program, for example, you do not have to define variables SHAPE, ROT, and SCALE. You always need XX, YY, and COLR to do anything meaningful, though.

What if you want to scale a shape, but not rotate it? You must define ROT anyway so that SCALE will end up in the correct position in the variable table. If you skip ROT and follow SHAPE with SCALE, the Apple system will interpret your SCALE variable as a bad attempt to rename ROT, and you will get an error message every time you run your program.

So it is important to define the hi-res variable with a certain number of characters and in a specific order. But how do you actually carry out the defining procedure? You do that by writing Integer BASIC program lines that equate those variable names to any value you wish.

Suppose that you will be using variables XX, YY, and COLR. Early in the BASIC program—before specifying any other variables—you can define them this way:

```
100 XX=0:YY=0 : COLR=0
```

Line 100 sets the variables equal to 0. You can equate them to any other value that strikes your fancy at the moment; it makes no difference what that value is.

Because of a quirk in Integer BASIC, you can simplify the definition procedure to this:

```
100 XX=YY=COLR
```

Or if you are planning to use all of the shape-table features:

```
100 XX=YY=COLR=SHAPE=ROT=SCALE
```

That initializes all six hi-res variables to the same value. There's no telling what that value is, but that isn't important anyway. Again, the important thing is to name the variables using the specified number of characters and placing the variables in the order shown in Table 7-3.

The idea is to make certain that those particular variables are placed at the very beginning of the variable table. Otherwise, the Programmer's Aid hi-res system won't be able to find them, because it always looks for them at the beginning of the variable table.

Recall from an earlier discussion that Integer BASIC variables are stacked from the current LOMEM setting and upward. If you insert a program statement such as A=25 before defining the special hi-res variables, that definition of variable A will be in the variable stack ahead of the hi-res variables and the scheme will blow up when you try to use them.

Now suppose that you want to POKE a LOMEM address of 3072 to the system before defining the hi-res variables. The programming in that case must begin this way:

```
100 POKE 74,0 : POKE 75,12
110 XX=YY=COLR=SHAPE=ROT=SCALE
```

Notice that you don't have to define the special hi-res variables before all other BASIC programming. You must, however, define those variables before specifying any other BASIC variables.

Most of the hi-res programs in this chapter begin with those two program lines. They set LOMEM to 3072 and define all six of the special hi-res variables.

**CALLing the INIT Routine** Recall that doing a GR statement brings up the normal, primary page of *low*-resolution graphics. The key to setting up the normal, primary page of *high*-resolution graphics is CALL-12288.

CALL -12288 from Integer BASIC initializes the normal high-resolution graphics mode.

CALLing that address instructs the system to execute a Programmer's Aid routine known as INIT. It is an important routine because it not only sets up the hi-res display, but also allows you to use the other hi-res features described in this chapter. Unless you CALL INIT, none of those special hi-res operations will work from Integer BASIC.

What is the normal hi-res graphics mode? It's like the low-res mode in most respects. That is, it leaves the four lower lines on the screen for text operations, it clears the graphics portion of the screen, and it displays the primary page of graphics.

There is one important difference, however: The INIT routine does not automatically set WNDTOP (the text window top) to the fourth line from the bottom of the screen. It also leaves the text page fully intact. That can be an advantage, of course, in situations where you wish to call up some hi-res graphics without disturbing primary-page text. But if you want to place new text under the hi-res graphics follow the CALL -12288 statement with this one:

```
POKE 34,20 : CALL -936
```

That sets the top of the scrolling window (RAM address 34) to 20, and clears the text portion of the screen.

Try it for yourself. Do a RESET followed by a CTRL-B to get into the command mode. Then enter a CALL -12288. That will bring up the normal high-resolution graphics. Then, if you wish, do a POKE 34,20 followed by a CALL -936 to clear the text portion of the display.

You cannot do anything further because this demonstration doesn't define the critical hi-res variables. But at least it shows what the high-resolution screen is like.

Enter TEXT to return from hi-res to the normal low-res text mode.

**The Complete Initialization Procedure** Simply CALLing INIT to get into the normal hi-res graphics mode is not enough. You must be able to plot some points or draw lines to make use of it. As mentioned earlier, you cannot do that without first defining at least three variables: XX, YY, and COLR.

So if INIT is going to be of any use, you must define the hi-res parameters. Consider this series of opening program lines:

---

```
100 POKE 74,0: POKE 75,12
110 XX=YY=COLR
120 CALL -12288
130 POKE 34,20: CALL -936
```

---

Line 100 sets LOMEM to address 3072. That step isn't necessary, however, if you are willing to use the default LOMEM of 2048 and you are certain that no previous operations have set it anywhere else.

Line 110 defines three variables for hi-res graphics. Those are the three that must always be used for hi-res graphics. If you plan to use the shape-table features, also, extend the line to include SHAPE, ROT, and SCALE.

Line 120 initializes the hi-res system by CALLing INIT, and line 130 sets the top of the text portion of the screen and homes the cursor within it.

**HIGH-RESOLUTION COLORS AND SCREEN FORMAT** Table 7-4 shows the six colors that are available for high-resolution graphics and their respective color codes. You can specify color codes from 0 to 255, but using any codes other than those shown here will cause the plotted points and lines to take on complex color combinations. (The actual colors created by codes might appear slightly different on your screen. The color rendition depends a great deal on the TINT and COLOR settings on your tv receiver or monitor.)

The normal high-resolution graphics screen allows four full lines of text at the bottom of the screen. The upper portion is set up according to the 280-by-160 format described in Fig. 7-1. Notice that there are 280 hori-

**Table 7-4. Hi-Res Color Codes**

Color	Code
BLACK	0 or 128
GREEN	42
VIOLET	85
WHITE	127 or 255
ORANGE	170
BLUE	213

zontal locations labeled 0 through 279, and 160 vertical locations labeled 0 through 159.

So there is a total of 44,800 hi-res plotting positions. You can access any one of them by assigning coordinates to the horizontal and vertical parameters, XX and YY. The general idea is similar to plotting points on the low-resolution graphics screen. There are simply more points involved here.

**DOING SOME HIGH-RESOLUTION GRAPHICS** It is possible to write some interesting and useful hi-res graphics programs without resorting to the use of special shape tables. Specifically, you can:

- Fill in a background color.
- Position a point without actually drawing it.
- Plot a single point.
- Plot a straight line.
- Clear the entire screen to black.

This section describes those high-resolution graphics operations.

When going through the discussions and demonstrations in this section, bear in mind that they must be preceded at some point by the hi-res initialization routines. The following lines must be written into all of the programs, but I will not be showing them each and every time. It's up to you to remember to insert these lines:

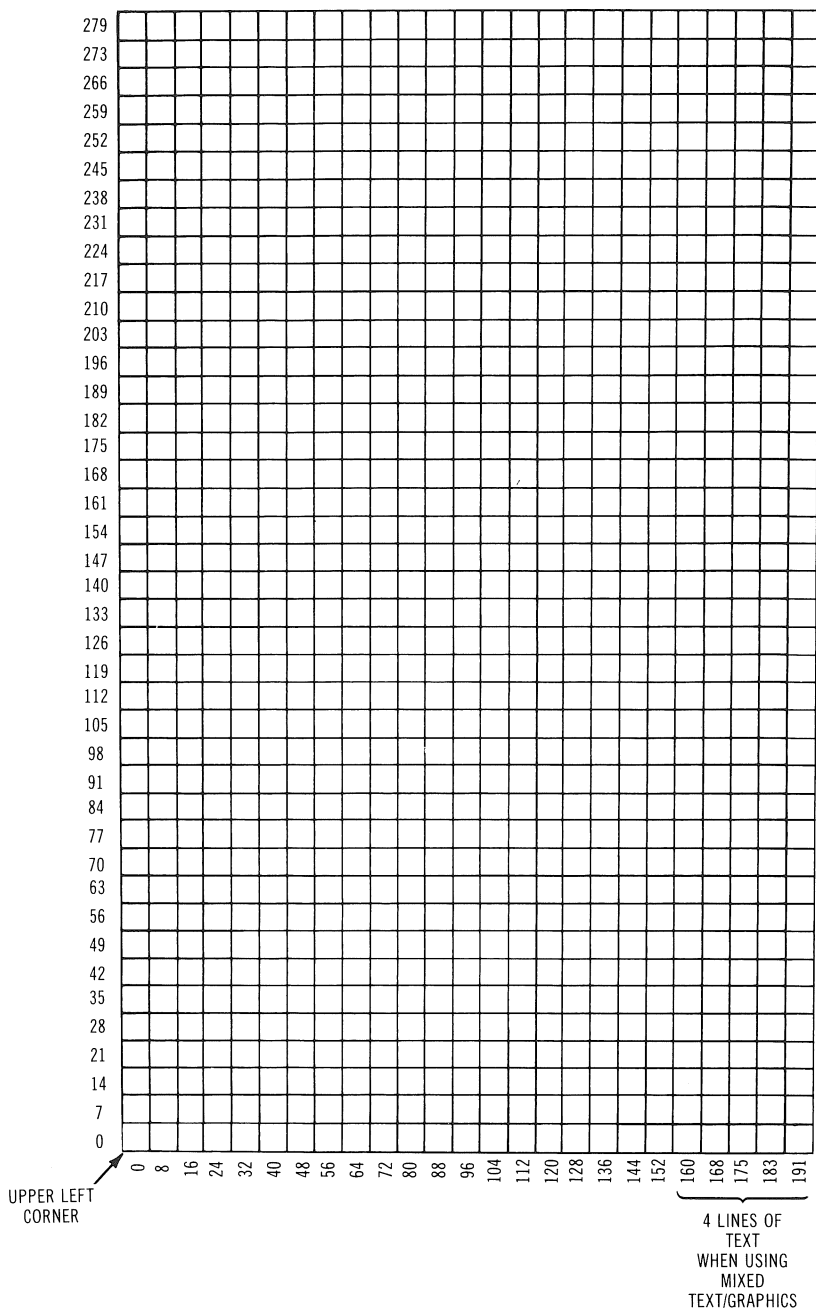
---

```
100 XX=YY=COLR
110 CALL -12288
```

---

**Filling in a Background Color** Calling the hi-res INIT routine automatically clears the graphics portion of the screen to black. Most of the time, however, you will want a somewhat more colorful background.

Setting up the background color is a two-step procedure. You must first specify the color and then call a background routine.



**Fig. 7-1. Hi-res graphics character locations.**

Specifying the color is a matter of equating the color parameter, COLR, to the desired background color from Table 7-4. So if you happen to want a green background, the appropriate color-specifying statement is COLR=42. But that simply defines the color. It is also necessary to call a Programmer's Aid routine, called BKGND, at -11471. You can execute it from BASIC by doing a CALL -11471.

CALL -11471 fills the hi-res screen with a color specified by a preceding COLR statement.

Enter and run this Integer BASIC demonstration program:

---

```
100 XX=YY=COLR
110 CALL -12288
120 COLR=42: CALL -11471
130 END
```

---

Lines 100 and 110 initialize the hi-res system, line 120 specifies a green color and calls the BKGND routine, and line 130 ends the program.

On running the program, you should see the entire low-res graphics portion of the screen filled with a color that appears more or less green (depending a lot on the tint setting of your TV monitor). You can, of course, specify other background colors by changing the value assigned to the COLR variable in line 120. Use the color codes recommended in Table 7-4 first, then try some of the other codes between 0 and 255. Maybe you will like some of those pretty striped patterns.

**Setting the Plot Coordinate** When you initialize the high-resolution graphics system, one of the first operations must be to set the plotting coordinate or the starting point of a line. Like most other hi-res operations, positioning a plotting point is a two-step procedure. First, you specify the desired XX and YY coordinates, and then, you call a point-positioning routine in the Programmer's Aid package.

Suppose that you want to begin with a plot position of 0,0—the extreme upper left-hand corner of the screen. You can specify that coordinate by entering XX=0 : YY=0. That is just the first of two steps, however. You must follow that statement by calling a routine known as POSN at address -11527. Thus a complete point-positioning routine looks like this:

```
XX=0 : YY=100 : CALL -11527
```

CALL -11527 sets the position of a point on the hi-res screen according to the preceding values of XX and YY.

That POSN routine is also used when you want to plot two different positions without having a line drawn between them. The following pair of BASIC program lines sets the system for plotting first at position 0,0 and then at 100,100:

```
XX=0 : YY=0 : CALL -11527
XX=100 : YY=100 : CALL -11527
```

The POSN routine creates no noticeable visual effect on the screen. It merely sets up the variables for a subsequent plotting operation.

**Plotting Points** Assuming that you have already written some program lines for initializing the hi-res system, you must carry out the following steps to plot a single point of hi-res color:

1. Position the point by setting up and executing the POSN routine.
2. Specify the desired plotting color by setting the COLR variable according to Table 7-4.
3. CALL the point-plotting routine, PLOT, at address -11506.

This program illustrates the entire plotting procedure. Give it a try by loading and running it.

---

```
100 XX=YY=COLR
110 CALL -12288
120 COLR=42: CALL -11471
130 XX=140:YY=80: CALL -11527
140 COLR=127: CALL -11506
150 END
```

---

Lines 100 and 110 initialize the hi-res graphics system, and line 120 fills in a green background. Line 130 sets the horizontal and vertical position of a point to coordinate 140,80, which is very close to the middle of the hi-res portion of the screen. Finally, line 140 sets the system for a white color and plots the color at the prescribed position.

Perhaps that seems like a lot of programming work just to plot a white dot in the middle of a green field, but that's the sort of demand that the high-resolution graphics scheme places on a programmer.

CALL -11506 plots a point on the high-resolution screen according to a color and position established earlier in the program.



Horizontal plotting of colors is tricky, because you can't plot one color at every horizontal coordinate. Table 7-5 shows the recommended color codes and their actual plotting colors. You can see that the horizontal XX position and background colors are critical.

Suppose for instance, that you want to plot some colored dots against a black background. You can see from the table that you can plot green points only at odd-numbered XX coordinates—1, 3, 5, and so on. Attempt to plot green (code 42) at some even-numbered XX location and you will see no response at all. (It would actually plot a black point against that black background, or worse, plot a black dot over some other color that might have been plotted there at an earlier time.)

Notice that using a black background offers the widest possible range of hi-res colors for plotting operations. It is also possible to plot white against a black background, but that is a two-step operation that we will describe a bit later in this discussion.

A white background renders only black plots, but at least they can be situated in any XX location. You can erase a previously drawn black point by overlaying it with a white plot.

The four remaining background colors support only black or white visible points and, even then, only in odd- or even-numbered XX positions. Also notice that code 127 is used for plotting white against green and violet backgrounds, while code 255 must be used for plotting white against orange and blue backgrounds. Depart from the format suggested in Table 7-5 and you will end up with some disappointing results.

Listing 7-1 is a program that lets you experiment with this notion of plotting colored points in odd- or even-numbered XX locations and against various background colors. If you intend to try any hi-res plotting of your own, it is important to get this experience ahead of time.

---

**Listing 7-1. Plotting Points.**

---

```
100 REM
110 XX=YY=COLR
120 CALL -12288
130 POKE 34,20: CALL -936
140 PRINT "WHAT BACKGROUND COLOR:"
150 INPUT "(SEE TABLE 7-5)",FIELD
160 PRINT "WHAT PLOT COLOR:"
170 INPUT "(SEE TABLE 7-5)",DOT
180 PRINT "WHAT XX LOCATION:"
190 INPUT "(0-279)",XX
200 COLR=FIELD: CALL -11471
210 YY=18: CALL -11527
220 COLR=DOT: CALL -11506
230 GOTO 140
```

---

**Table 7-5. Hi-Res Color Codes With Different Background Colors**

<b>Background Color</b>	<b>Hi-Res Color Code</b>	<b>Hi-Res Color</b>
BLACK (0)	42	GREEN at odd-numbered XX locations BLACK at even-numbered XX locations
	85	VIOLET at even-numbered XX locations BLACK at odd-numbered XX locations
	170	ORANGE at odd-numbered XX locations BLACK at even-numbered XX locations
	213	BLUE at even-numbered XX locations BLACK at odd-numbered XX locations
	0	BLACK at all XX locations
WHITE (255)	0	BLACK at all XX locations
	255	WHITE at all XX locations
GREEN (42)	0	BLACK at all odd-numbered XX locations GREEN at all even-numbered XX locations
	127	WHITE at all even-numbered XX locations GREEN at all odd-numbered XX locations
VIOLET (85)	0	BLACK at all even-numbered XX locations VIOLET at all odd-numbered XX locations
	127	WHITE at all odd-numbered XX locations VIOLET at all even-numbered XX locations
ORANGE (170)	128	BLACK at all odd-numbered locations ORANGE at all even-numbered locations
	255	WHITE at all even-numbered XX locations ORANGE at all odd-numbered XX locations
BLUE (213)	128	BLACK at all even-numbered XX locations BLUE at all odd-numbered XX locations
	255	WHITE at all odd-numbered XX locations BLUE at all even-numbered XX locations

Enter the program, run it, and respond to the prompt messages that request a background color code, a plotting color code, and the XX component of the plotting position. (The program sets YY to a value of 18 in every case.)

As long as you adhere to the values and limitations specified in Table 7-5, you will get the desired results. Depart from them, and you will get dots of the wrong color, multicolored strokes of light, or nothing at all.

Here is a detailed analysis of how the program works. You can use it as a review of matters discussed thus far.

Lines 110 and 120 define the hi-res variables and call the INIT routine.

Line 130 sets the top of the text window, homes the cursor, and clears the text screen.

Lines 140 and 150 input the desired background color code as variable FIELD.

Lines 160 and 170 input the desired plotting color code as variable DOT.

Lines 180 and 190 input the desired XX component of the plotting position.

Line 200 fills the background with color FIELD.

Line 210 establishes the XX,YY position by calling the POSN routine.

Line 220 sets the dot color and calls the PLOT routine.

Line 230 returns to line 140 to start specifying another combination of background color, dot color, and XX position.

Listing 7-2 uses these principles to create an interesting graphic—200 randomly positioned white dots against a blue background.

---

**Listing 7-2. Random Points.**

---

```
100 XX=YY=COLR
110 CALL -12288
120 POKE 34,20: CALL -936
130 TAB 12
140 PRINT "*** STARRY SKY ***"
150 COLR=213: CALL -11471
160 COLR=255
170 FOR N=0 TO 199
180 XX= RND (279):YY= RND (179)
190 CALL -11527
200 CALL -11506
210 NEXT N
220 END
```

---

See if you can justify every step in that program.

It is possible to plot white dots onto a black background, but only by using a two-step plotting procedure. Given a black background, the idea is

to plot a 127 WHITE color code at any horizontal XX position and another at position XX+1. In other words, plot two 127s in successive XX locations. The YY components of the coordinates are not relevant, but must be equal.

Some BASIC programming for doing the job uses this series of statements:

```
COLR=127
```

```
XX=100 : YY=80 : CALL -11527 : CALL -11506
```

```
XX=XX+1 : CALL -11527 : CALL -11506
```

Try developing some simple hi-res programs of your own.

**Drawing Straight Lines** It is possible to draw straight horizontal or vertical lines of some prescribed color by doing a long series of PLOTs to an equally long series of XX,YY coordinates. Not only is that a tedious programming task, but it is also a slow drawing operation.

The alternative is to use a routine built into the Programmer's Aid called LINE. You can execute a properly set up LINE operation by CALLing -11500.

CALL -11500 draws a line of some prescribed color between two previously established endpoint coordinates.

In principle, executing the LINE routine is a three-step procedure:

1. Establish the XX,YY coordinate of the beginning of the line and do a POSN operation.
2. Specify the XX,YY coordinate of the end of the line.
3. Specify a color and CALL the LINE routine.

You can establish the starting point of the line with a BASIC statement such as:

```
XX=0:YY=100 : CALL -11527
```

That program line fixes the starting coordinate of the line at 0,100 and calls the POSN routine to fix those coordinates in memory.

As an example of the second line-drawing step, consider this:

```
XX=100 : YY=120
```

That establishes the final coordinate of the line at 100,120. *Do not CALL the POSN routine after specifying the final coordinate of the line.* If you do that, you will be resetting the starting point.

Finally, you can set the color and call the LINE routine with:

**COLR=255:CALL -11527**

That series of steps will draw a more or less straight line between coordinates 0,100 and 100,120. Here is a complete programming routine for drawing a white version of that line against an orange background.

---

```
100 XX=YY=COLR
110 CALL -12288
120 COLR=170: CALL -11471
130 XX=0:YY=100: CALL -11527
140 XX=100:YY=120
150 COLR=255: CALL -11500
160 END
```

---

Program lines 100 and 110 initialize the hi-res system, and line 120 sets the background color and calls the BKGND routine to fill in the field with that ORANGE color. Line 130 is actually the first step in the line-drawing procedure. It sets up the starting coordinate and executes the POSN routine. Line 140 then sets the end-of-line coordinate, while line 150 sets the line color to WHITE and draws the line by calling the LINE routine.

There are a couple of ideas that will help simplify hi-res programming routines. The first one is, if you do not specify an XX or YY component for the end of the line, the system will default to the last-specified value. The second idea is that executing LINE performs the equivalent of a POSN routine using the coordinates specified for the end of the line. The significance of the latter idea is that you need not use POSN to set the coordinate for the next point or line if it is to be situated at the end of a line you've just drawn. Consider the following program that draws a white square onto a green background.

---

```
100 XX=YY=COLR
110 CALL -12288
120 COLR=42: CALL -11471
130 COLR=127
140 XX=120:YY=60: CALL -11527
150 XX=160: CALL -11500
160 XX=100: CALL -11500
170 XX=120: CALL -11500
180 YY=60: CALL -11500
190 END
```

---

Here is how it works:

Lines 100 and 110 define the critical hi-res variables and initialize the system.

Line 120 sets up and draws a green background.

Line 130 defines a WHITE color for the GREEN background.

Line 140 establishes the initial coordinate of the drawing by calling the POSN routine.

Line 150 draws the first line between 120,60 and 160,60. (The default value for the YY component is 60 from the previous program line.)

Line 160 draws the second line between 160,60 and 160,100. (The default value for the XX component is 160 from the previous operation.)

Line 170 draws the third line from coordinate 160,100 to 120,100.

Line 180 draws the final line from coordinate 120,100 to 120,60.

You can change the background color by altering the COLR assignment in program line 120, and you can specify other line colors in program line 130. Use Table 7-4 as a guide for selecting the colors. See if you can modify the program to draw a black square on a white background, for example.

Listing 7-3 is a program that lets you experiment with drawing single lines of any chosen color between any chosen sets of coordinates. It also lets you select a background color.

---

#### **Listing 7-3. Drawing Lines.**

---

```
100 XX=YY=COLR
110 TEXT : CALL -936
120 PRINT "WHAT BACKGROUND COLOR:"
130 INPUT "(SEE TABLE 7-4)",BCOLR
140 INPUT "WHAT STARTING XX (0-279)",XS
150 INPUT "WHAT STARTING YY (0-159)",YS
160 INPUT "WHAT ENDING XX (0-279)",XE
170 INPUT "WHAT ENDING YY (0-159)",YE
180 PRINT "WHAT LINE COLOR:"
190 INPUT "(SEE TABLE 7-4)",LCOLR
200 CALL -12288
210 POKE 34,20: CALL -936
220 COLR=BCOLR: CALL -11471
230 XX=XS:YY=YS: CALL -11527
240 XX=XE:YY=YE
250 COLR=LCOLR: CALL -11500
260 PRINT
270 PRINT "STRIKE ANY KEY TO DO AGAIN ..."
280 CALL -741
290 GOTO 110
```

---

Let us study the operation of this demonstration program:

Line 100 defines the critical hi-res variables.

Line 110 brings the system into the TEXT mode, homes the cursor, and clears the screen.

Lines 120 and 130 ask for the background color code.

Lines 140 and 150 ask for the starting coordinates of the line.

Lines 160 and 170 ask for the ending coordinates of the line.

Lines 180 and 190 ask for the color code of the line.

Line 200 brings the system into the hi-res graphics mode by calling the INIT routine.

Line 210 sets the top of the text window, homes the cursor, and clears the screen.

Line 220 fills the background with color BCOLR.

Line 230 sets the starting coordinates to the values typed in earlier by the user, and then CALLs the POSN routine.

Line 240 sets the ending coordinates to the values typed in earlier by the user.

Line 250 sets the line color to the value typed in earlier by the user, and then CALLs the LINE routine.

The remainder of the program simply prints a prompt message in the text portion of the screen and waits for you to strike any key to do the whole thing all over again.

You have probably noticed by now the hi-res, line-drawing routine is something less than perfect. Almost without exception, there is some color distortion at the beginning and end of any line. You will find that you can plot any line color onto any background color as long as the line is perfectly horizontal. But add a vertical component to the slope of the line, and you will find some color distortion along the line as well as at the ends. The worst distortion occurs when plotting a straight vertical line.

It is possible to get into some very heavy technical discussions of how and why this whole family of color-distortion problems occurs and how to remedy some of them. But even then, the complexity of the techniques do not justify the less-than-satisfactory results.

In the most practical sense, the matter is best handled on a trial-and-error basis for each particular case. If you find that the color distortion is intolerable in a particular instance, try shifting the coordinates one XX location to the left or right. That might not cure the problem altogether, but it can produce more satisfactory results in many instances.

Another trick is to plot the line and then plot discrete points at the places where the most serious color distortion occurs.

**Clearing the Hi-Res Screen** The Programmer's Aid package includes a routine that clears the hi-res screen to all black. CALLing INIT does that, too, but only at the beginning of the program. The special screen-clearing routine, called CLEAR, begins at address -12274.

CALL -12274 clears the high-resolution portion of the screen to all black.

**Simplifying the CALLs** By now you are aware of an almost overwhelming number of CALL routines for running hi-res graphics programs. Having to remember or look up those CALL addresses every time you use one of them can be troublesome and a source of possible programming errors. So it's a good idea to set those CALL addresses equal to some meaningful variable names at some place near the beginning of the program—certainly after specifying the critical hi-res variables XX, YY, and COLR. Here is an example:

---

```
100 XX=YY=COLR
110 INIT=-12288:BKGND=-11471
120 POSN=-11527:PLOT=-11506:LINE=-11500
130 CLEAR=-12274
```

---

After doing that, you can initialize the hi-res system by executing a CALL INIT, call the BKGND routine by executing a call BKGND, set a screen position by executing a CALL POSN, and so on. The idea is to call the main hi-res routines by name rather than number.

Listing 7-4 uses that technique for POSN, PLOT, and LINE. It doesn't apply the idea to INIT and BKGND because they are used only once in the entire program. Also notice in lines 130 through 150 that it is possible to make similar assignments for the color codes.

Enter the program and run it. Study the listing to get some ideas about drawing blocks of color with the LINE routine and overcoming certain kinds of color distortion.



#### Listing 7-4. Winter Scene.

---

```
100 REM
110 XX=YY=COLR
120 POSN=-11527:PLOT=-11506:LINE=-11500
130 BLACK=0: LET GREEN=42:VIOLET=85
140 WHITE=127:ORANGE=170:BLUE=213
150 WHITEH=255
160 POKE 34,20: CALL -936
170 CALL -12288
180 FOR PH=1 TO 3: GOSUB PH*1000: NEXT PH
190 PRINT : PRINT : TAB 10
200 PRINT "*** WINTER SCENE ***"
210 PRINT : END
1000 REM ** PLOT BACKGROUND **
1010 COLR=BLUE: CALL -11471
1020 COLR=WHITEH
1030 FOR N=1 TO 300
1040 XX= RND (279):YY= RND (159)
1050 CALL POSN: CALL PLOT
1060 NEXT N
1070 RETURN
2000 REM ** DRAW SNOW **
2010 COLR=WHITEH
2020 FOR N=0 TO 24
2030 XX=0:YY=135+N: CALL POSN
2040 XX=279: CALL LINE
2050 NEXT N
2060 RETURN
3000 REM ** TREE **
3010 COLR=WHITEH
3020 FOR N=0 TO 99
3030 XX=140-N/2:YY=10+N: CALL POSN
3040 XX=140+N/2: CALL LINE
3050 NEXT N
3060 COLR=GREEN
3070 FOR N=7 TO 99
3080 XX=144-N/2:YY=10+N: CALL POSN
3090 XX=136+N/2: CALL LINE
3100 NEXT N
3110 COLR=BLACK
3120 FOR N=0 TO 30
3130 XX=140-3:YY=110+N: CALL POSN
3140 XX=140+5: CALL LINE
3150 NEXT N
3160 COLR=BLUE
3170 FOR N=0 TO 24
3180 XX=140-5:YY=110+N: CALL PLOT
3190 NEXT N
3200 RETURN
```

---

**ALTERNATIVE HI-RES SCREEN FORMATS** There are two alternatives to the normal high-resolution graphics screen format: full-screen graphics and the secondary hi-res screen.

**Full-Screen Graphics** The normal high-resolution graphics scheme, called INIT, uses the primary page with mixed text and graphics. The range of XX and YY coordinates is from 0 to 279, and 0 to 159, respectively. The four lower, text-sized lines are open for text operations.

Going to full-screen, hi-res graphics deletes the four lower lines of text and opens them for the graphics operations. The screen format in that case allows 280 horizontal XX positions, and 192 vertical YY positions.

There are several steps involved in setting up full-screen hi-res graphics. Basically they amount to POKEing zeros into the screen “soft” switch positions described earlier. Table 7-6 summarizes the list of “soft” switches as extended to include some hi-res addresses.

Suppose, then, that you want to use a full screen of high-resolution graphics. According to Table 7-6 an appropriate sequence of POKE statements would be:

POKE -16297,0 (Hi-res.)  
POKE -16300,0 (Primary page.)  
POKE -16302,0 (All graphics.)  
POKE -16304,0 (Graphics mode.)

Try that sequence of POKES from the keyboard, and you will see a full screen of hi-res graphics.

CALLing INIT to set up mixed-screen graphics automatically clears the graphics portion of the screen to black. This POKE sequence does not;

**Table 7-6. Graphics Soft Switches**

Graphics Mode	POKE Address
Display high-resolution graphics	-16297
Display low-resolution graphics	-16298
Display the secondary page	-16299
Display the primary page	-16300
Display mixed text and graphics	-16301
Display all text or all graphics	-16302
Display a text mode	-16303
Display a graphics mode	-16304

so you might want to conclude it with a CALL -12274, which is the clear-to-black routine.

Here are some program lines that will get you started with full-screen hi-res. Add your own ideas about plotting lines and points, remembering that you can now extend the YY values to 191.

---

```
100 XX=YY=COLR
110 POKE -16297,0: POKE -16300,0
120 POKE -16302,0: POKE -16304,0
130 CALL -12274
```

---

Use TEXT to return to the normal, all-text mode.

**Secondary-Page Graphics** If you have a 16K system, this discussion is purely academic—you have no secondary page of hi-res graphics available to you. Otherwise you may be in for a somewhat pleasant surprise: The secondary page of hi-res graphics is actually simpler to use than the secondary page of lo-res graphics.

One of the truly positive features of the hi-res system is that you can plot points and draw lines on the secondary page while displaying the primary page, and *vice versa*. Recall that it is impossible to draw directly onto the secondary page of low-res graphics.

The key to choosing the page that will get the result of hi-res programming is memory address 806. Doing a POKE 806,32 will cause any hi-res drawing operations to take place on the primary page of graphics, no matter which page is being displayed at the time. On the other hand, doing a POKE 806,64 will cause drawing to take place directly on the secondary page. Again, it makes no difference which is being displayed at the time.

POKE 806,32 allows you to draw on the primary page of hi-res.

POKE 806,64 allows you to draw on the secondary page of hi-res.

Drawing high-resolution pictures can, in many instances, be a relatively time-consuming process. Being able to draw on one page while displaying the other lets you mask the long drawing times by entertaining the viewer with a finished picture on the other page. Or you can display some meaningful text while the program is drawing some high-resolution graphics on a “hidden” page. Then, when the drawing is done, simply replace the text presentation with the hi-res picture. The viewer at least gets the impression that the drawing operation took place instantaneously.

**HI-RES SHAPE TABLES** Shape tables are a bit difficult (or at least tedious) to design and enter into the system; but you ought to give them serious consideration because of two advantages they have.

One advantage is that the Apple hi-res graphics system can draw figures from tables a whole lot faster than it can draw from lists of PLOT and LINE statements. This feature is especially important when attempting to write programs that produce satisfactory animation sequences.

A second advantage is that the Apple system includes provisions for scaling and rotating graphics produced from shape tables. So if you have a graphic shape that you want to scale (change size) or rotate (turn about a given point), then shape tables are your best bet.

High-speed drawing and the ability to scale and rotate a figure are the positive attributes of shape-table graphics. The only negative attribute is that they work in a fashion more akin to machine-language programming than to BASIC. It's not a bad scheme; it simply appears unfamiliar and awkward to anyone unaccustomed to machine-language techniques.

**What Is a Shape Table?** From one point of view, a shape table is a block of RAM that is set aside for data related to a particular graphics shape or, indeed, as many as 255 different graphics shapes. Whenever a BASIC program executes a DRAW or DRAW1 command, the system consults the shape table for the appropriate drawing information. The hi-res ROT, SCALE, and FIND commands also refer to the shape table.

One of the preliminary steps in implementing shape tables is setting up that block of RAM—specifying its starting address, for instance.

The table, itself, is divided into two main parts: the shape table *index* and the shape data for each figure. A single index serves from 1 to 255 different shapes.

The index is simply a group of numbers that indicates the number of different shapes in the table, and the number of address locations from the beginning of the index to the beginning of the data for each shape. The size of the index—that is, the number of codes in it—depends on the number of different shapes in the table. The larger the number of shapes, the larger the index.

The shape data for each figure consists of a set of codes that tells the system to plot on or skip over a hi-res point on the screen. And equally important, the data specifies where an imaginary hi-res cursor should move to after doing the previous plotting or no-plotting operation. Finally, a code-number 0 marks the end of the block of data for each shape in the table.

The length of the data block for each shape in the table depends on how complicated and how large the shape is. Generally speaking, the more complex and larger the shape, the longer its data block is.

**Setting Up Shapes From BASIC** As mentioned earlier, the Apple draws shapes from the shape table via BASIC routines such as DRAW and DRAW1. But it is important to set up some other parameters in advance.

The first parameter is the screen position of the figure to be drawn from the table. That parameter is specified by assigning position values to the XX and YY variables, and then executing a POSN routine. That sets the starting position for drawing a shape. Actually, any technique that works for PLOT and LINE operations works equally well for shape tables. Additionally, the DRAW1 routine lets you begin drawing a second shape at the point on the screen where the system completed drawing the first shape.

The second parameter is the shape color. The color codes used are the same as those summarized in Table 7-5. For a beginner, the safest all-around color combination is a white shape on a black background. The actual color of the shape depends on its horizontal screen position and the way you have arranged the plot and no-plot sequences within the shape table.

The third parameter is the values for SCALE and ROT. Even if you don't plan to scale or rotate the shape from its configuration specified in the shape table, you must define those variables early in the program and set them equal to 0 at some point prior to calling up the DRAW or DRAW1 routine.

The final parameter is the shape number. The shapes in your table must be specified with numeric values 1 through n, where n is the final shape in the table. The variable required here is SHAPE. So including a statement such as SHAPE=4 designates the fourth block of shape data in the table. Of course you need four or more shapes in the table to use that particular example. If you are playing around with just one shape, you must include a SHAPE=1 before calling the routines that scale, rotate, and eventually draw the shape on the screen.

The shape table is loaded by means of POKE statements. This technique is quite different from the one suggested in most of the Apple literature. There, you are expected to enter the shape table as hexadecimal or binary data files. The procedure offered here accomplishes the same goal from a purely BASIC approach. To be sure, entering long strings of POKE statements can be tiresome, but once the task is done, you can save and reload the shape table and BASIC programming as a single BASIC program.

It all might seem quite complicated and confusing at first. Perhaps it is. But the whole thing can become rather routine after working with it for a while. Besides, this is simply a preview. The remaining discussions in this chapter are more detailed.

**Preparing Data for the Shape Table** The data block for a particular shape consists of code numbers, each one indicating the following:

1. Whether or not to plot at the current hi-res point on the screen.
2. Where to move on the screen after doing that plot or no-plot operation.

The shape-table data carries no direct color information. That is set by the COLR variable in the BASIC programming. All the shape table does is indicate whether or not to plot a point, and then where to go from there.

Table 7-7 summarizes the most useful shape table codes. There are many more possibilities, but these are the ones that cause the least amount of confusion for people who are not already fully familiar with the scheme.

Notice in each case that it is possible to plot or not plot. Then there is a direction of motion: up, down, right, or left. The general idea is to plot or not plot, and then move away in one of those four directions.

Some instances allow multiple operations of the same kind. A code 36, for example, does a plot-move combination twice in succession. It is the same as doing two separate code-4 operations. And in three nonplotting

**Table 7-7. Shape Table Acronyms and Codes**

Acronym	Code	Meaning
1UN	128	Don't plot; move 1 space upward
1UP	4	Plot 1 space upward
2UP	36	Plot 2 spaces upward
1DN	2	Don't plot; move 1 space downward
2DN	18	Don't plot; move 2 spaces downward
3DN	146	Don't plot; move 3 spaces downward
1DP	6	Plot 1 space downward
2DP	54	Plot 2 spaces downward
1RN	1	Don't plot; move 1 space to right
2RN	9	Don't plot; move 2 spaces to right
3RN	73	Don't plot; move 3 spaces to right
1RP	5	Plot 1 space to right
2RP	45	Plot 2 spaces to right
1LN	3	Don't plot; move 1 space to left
2LN	27	Don't plot; move 2 spaces to left
3LN	129	Don't plot; move 3 spaces to left
1LP	7	Plot 1 space to left
2LP	63	Plot 2 spaces to left

instances, a single code number can move three consecutive spaces. Code 219, as an example, moves three points to the left without plotting anything along the way.

The first step in preparing a shape table is to draw the desired figure on a sheet of graph paper, allowing each square to represent one hi-res point on the screen.

With a satisfactory figure thus drawn, pick a starting point and build a list of shape-table codes, using Table 7-7 as a guide. Bear in mind that the computer will draw the figure in the same sequence that you list the codes. Finally, you must end the block of data for each shape with a code 0. If you are using more than one shape in the table, each must end with a 0. The system uses the 0 to know when it is time to stop drawing a particular shape and return to the BASIC controlling routine.

If you are working with more than one shape, you will find it helpful in the early going to count the number of codes, including the end-marking 0s, in each shape block. Keep track of those numbers for the time when you are preparing the index.

Suppose that you want to draw the square figure in Fig. 7-2. Draw the figure onto a sheet of graph paper, letting each square represent one hi-res screen location. We aren't interested in the actual screen location, just the position of each point relative to the chosen starting location.

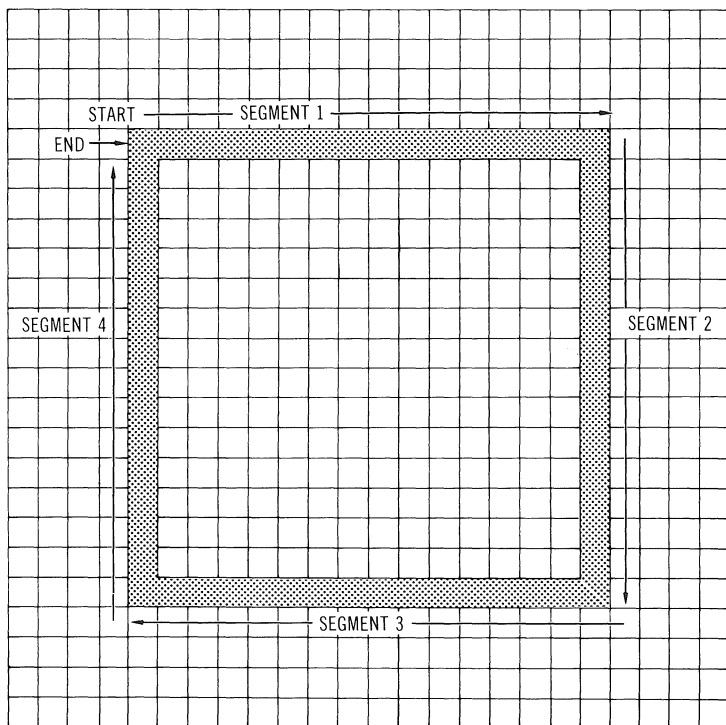
In this particular case, we begin the shape in the upper left-hand corner, proceed to the right along the top, go down the right-hand side, go to the left along the bottom, and finally return to the starting point by moving upward along the left-hand side of the figure. You can draw such a figure in any sequence you like, and begin and end anywhere you choose. The approach we use here simply seems to be the most direct one in this instance.

At this stage of the operation, it is easier to think in terms of acronyms than in code numbers. So the first step in the analysis is to represent the drawing sequence in shorthand form: 2RP (plot two spaces to the right), 2RP (plot two spaces to the right), and so on around the figure. Note especially how we deal with the corners.

Once you are satisfied with the sequence of acronyms, simply use Table 7-7 to assign the corresponding code numbers. End the sequence with a 0, count the number codes (including the end-marking 0), and the table designing task is done.

Figure 7-3 shows a little character that might be familiar to most arcade game aficionados. It represents a somewhat more complex figure than the previous one, but its shape table lends itself to a more systematic preparation.

In this particular instance, it is easier to divide the shape into vertical segments—S1, S2, S3, and so on. Beginning at the top of segment S1, we generate the acronyms for drawing that segment. The segment ends by plotting a single point and moving one space to the right. That brings us to



Segment 1 →	Segment 2 ↓	Segment 3 ←	Segment 4 ↑
2RP 45	2DP 54	2LP 63	2UP 36
2RP 45	2DP 54	2LP 63	2UP 36
2RP 45	2DP 54	2LP 63	2UP 36
2RP 45	2DP 54	2LP 63	2UP 36
2RP 45	2DP 54	2LP 63	2UP 36
2RP 45	2DP 54	2LP 63	2UP 36
2RP 45	2DP 54	2LP 63	2UP 36
2RP 45	2DP 54	2LP 63	2UP 36
1RP 5	1DP 6	1LP 7	1UP 4
			END 0

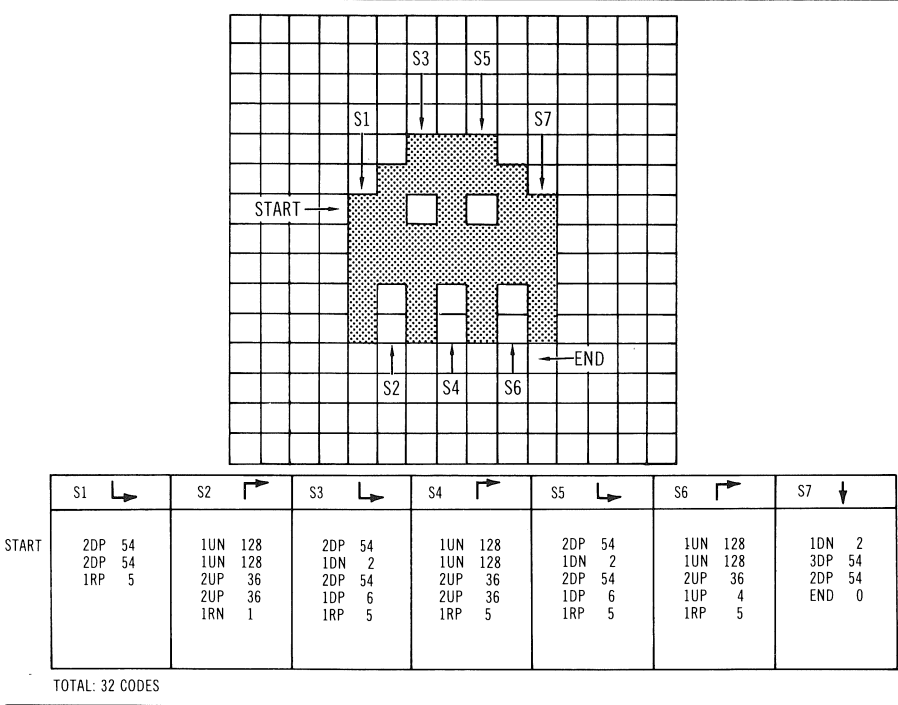
TOTAL: 33 CODES

Fig. 7-2. Hi-res rectangle drawing sequence.

the bottom of segment S2 and the starting point for drawing that segment from bottom to top. We continue the procedure until we reach the final point at the bottom of segment S7.

We could have organized the shape into horizontal segments, but they would have required more codes. Some shapes, however, lend themselves to that horizontal “scanning” approach.





**Fig. 7-3. Hi-res complex figure drawing sequence.**

No matter how you approach the drawing sequence, you should end up with a series of code numbers that concludes with a 0. That is the data block for a given shape.

**Completing the Shape Table With the Index** Whether you are using a single shape or a hundred of them, the shape table must begin with a single index that indicates (1) the number of shapes in the table, and (2) the starting position of each block of shape data.

An index for one shape is the simplest index. It contains four codes: one code to indicate the number of shapes, an irrelevant number, and two codes to indicate the starting position of the shape data. An index for two shapes contains six codes: one code to indicate the number of shapes, that irrelevant number again, and two pairs of codes indicating the starting positions of the two shape-data blocks. An index for three shapes has eight codes: one for the number of shapes, the irrelevant number, and three pairs of codes pointing to the beginning of each shape-data block.

Clearly, the size of the index depends on the number of different shapes specified in the shape table. It is always equal to twice the number of shapes plus two.

Assuming that you have already prepared shape data, take a sheet of ordinary notebook paper and label each line with integer values from 0 to twice the number of shapes plus two. Then continue labeling to some number you figure will be adequate for listing all of the shape codes. (You can always add or delete lines at the end.)

Line 0 marks the beginning of the index. You should know how many shapes you are using, so write that number on line 0 of the chart. Write any number you like on line 1. Then skip two lines for every shape in the table. If you are using two shapes, skip two lines, if you have three shapes, skip 6 lines, and so on. That line—whatever it may be—marks the end of the index. The shape data for the first figure begins on the line that follows.

Suppose that you are building a table for two different shapes. That being the case, the index portion starts out looking something like Fig. 7-4.

---

LINE	CODE	NOTES
0	2	START OF INDEX
1	0	
2		
3		
4		
5		
6		
7		
8		
9		

---

Fig. 7-4. Shape table worksheet.

Begin writing in the sequences of shape data. The first shape you enter will be called shape 1 from then on, the second shape will be designated shape 2, and so on. Continue entering the shape data until you get them all into place on the worksheet.

Table 7-8 shows a shape table worksheet that includes the shape data from Figs. 7-2 and 7-3. The rectangle is shape 1, and the little creature is shape 2.

The table occupies lines 0 through 70. All that remains to be done is to assign the shape starting codes to lines 2 through 5.

Now notice that shape 1 begins at line 6 and that shape 2 begins at line 39. Those are the numbers that are important to the index. They must be entered as 2-byte decimal numbers, however. (See Appendix A.) That

**Table 7-8. Shape Table Worksheet**

Line	Code	Table Section
0	2	PARTIAL INDEX FOR 2 SHAPES
1	0	
2		
3		
4		
5		
6	45	SHAPE 1 DATA
7	45	
8	45	
9	45	
10	45	
11	45	
12	45	
13	5	
14	54	
15	54	
16	54	
17	54	
18	54	
19	54	
20	54	
21	6	
22	63	
23	63	
24	63	
25	63	
26	63	
27	63	
28	63	
29	7	
30	36	
31	36	
32	36	
33	36	
34	36	
35	36	
36	36	
37	4	
38	0	

Table 7-8—cont. Shape Table Worksheet

Line	Code	Table Section
39	54	SHAPE 2 DATA
40	54	
41	5	
42	128	
43	128	
44	36	
45	36	
46	1	
47	54	
48	2	
49	54	
50	6	
51	5	
52	128	
53	128	
54	6	
55	36	
56	5	
57	54	
58	2	
59	54	
60	6	
61	5	
62	128	
63	128	
64	36	
65	4	
66	5	
67	2	
68	54	
69	54	
70	0	

Table 7-9. Completed Shape Index

LINE	CODE
0	2
1	0
2	6 } LINES TO SHAPE 1
3	0 }
4	39 } LINES TO SHAPE 2
5	0 }

means the codes for shape 1 are 6 and 0; and the codes for shape 2 are 39 and 0.

So write 6 and 0 into lines 2 and 3, respectively; and write 39 and 0 into lines 4 and 5, respectively. That completes all the work for the shape table worksheet. Table 7-9 shows the completed shape index.

You have seen how to prepare shape tables. The next step is to write a BASIC program that loads the table into your Apple system.

**Writing the BASIC Shape-Table Loader** The shape table that you've prepared must be loaded into a block of RAM. At this point, the actual RAM addresses aren't important. In fact, you should write the BASIC loader routine in such a way that you can place the shape table anywhere you choose. The basic idea is to prepare a series of POKE statements that will POKE the table into successive address locations.

Listing 7-5 represents the BASIC shape-table loader for the table illustrated in Table 7-9. Variable ST represents the starting address of the shape table. It isn't defined in this BASIC routine, but you will eventually assign some value to it in a program that calls this subroutine. Do not attempt to run this subroutine as shown here—that comes later.

Referring to Table 7-9, you can see that program lines 1010 through 1030 POKE the index into consecutive address locations ST+0 through ST+5. Throughout this program, the number summed with variable ST is equal to the line number from the shape-table worksheet.

Program lines 1050 through 1120 load the shape data for shape 1. Since there are four instances where the table calls for loading the same number seven times in succession, using FOR-NEXT loops helps simplify the programming procedure. Loading shape 2 is not quite so simple, however.

Program lines 1140 through 1270 are responsible for loading the data for shape 2. The technique used here might seem terribly cumbersome at first glance, but after you see how it works, you will see that it is simpler than typing in 32 individual POKE statements.

The idea is to set variable N in a FOR-NEXT loop that covers the entire range of worksheet line numbers for shape 2—lines 39 through 70, in this case. The loop includes a GOSUB to a small (a very small) subroutine that assigns the current data value to variable K. On returning from the subroutine, K is POKEd into the proper memory location for the shape table.

The short subroutines occupying program lines 1239 through 1270 represent the line-by-line shape data for shape 2. To get those lines into the program, do an AUTO 1239,1 and begin typing them in. It's easier than it looks. And it is certainly easier than having to type in 32 separate POKE statements.

## Listing 7-5. Shape-Table Loader.

---

```
1000 REM      ** LOAD SHAPE TABLE **
1010 POKE ST+0,2: POKE ST+1,0
1020 POKE ST+2,6: POKE ST+3,0
1030 POKE ST+4,39: POKE ST+5,0
1040 REM      ** SHAPE 1 **
1050 FOR N=6 TO 12: POKE ST+N,45: NEXT N
1060 POKE ST+13,5
1070 FOR N=14 TO 20: POKE ST+N,54: NEXT N
1080 POKE ST+21,6
1090 FOR N=22 TO 28: POKE ST+N,63: NEXT N
1100 POKE ST+29,7
1110 FOR N=30 TO 36: POKE ST+N,36: NEXT N
1120 POKE ST+37,4: POKE ST+38,0
1130 REM      ** SHAPE 2 **
1140 FOR N=39 TO 70: GOSUB 1200+N
1150 POKE ST+N,K: NEXT N
1160 RETURN
1170 REM
1180 REM
1239 K=54: RETURN
1240 K=54: RETURN
1241 K=5: RETURN
1242 K=128: RETURN
1243 K=128: RETURN
1244 K=36: RETURN
1245 K=36: RETURN
1246 K=1: RETURN
1247 K=54: RETURN
1248 K=2: RETURN
1249 K=54: RETURN
1250 K=6: RETURN
1251 K=5: RETURN
1252 K=128: RETURN
1253 K=128: RETURN
1254 K=36: RETURN
1255 K=36: RETURN
1256 K=5: RETURN
1257 K=54: RETURN
1258 K=2: RETURN
1259 K=54: RETURN
1260 K=6: RETURN
1261 K=5: RETURN
1262 K=128: RETURN
1263 K=128: RETURN
1264 K=36: RETURN
1265 K=4: RETURN
1266 K=5: RETURN
1267 K=2: RETURN
1268 K=54: RETURN
1269 K=54: RETURN
1270 K=0: RETURN
```

Overall, the idea is to POKE the shape table into RAM, beginning at address ST. Use any programming procedure that suits you.

If you want to test the operation of this BASIC loader, first do a LOMEM:3072 and then type in this short routine:

---

```
100 ST=2048: GOSUB 1000
110 FOR N=0 TO 70
130 PRINT PEEK (ST+N); "/"
140 NEXT N
150 END
```

---

It is important to do the LOMEM:3072 to raise system LOMEM to that point, thereby leaving some room in the lower RAM area for the shape table. Line 100 in the test program assigns address 2048 as the starting address of the table, and then it calls the loader routine to POKE the table into that area.

Lines 110 through 140 simply PEEK into the table, printing out the data in the sequence it appears on your table worksheet. The slash simply separates one code from the next.

Delete the test program after you've had a chance to check out the operation of the table loading routine and the data. It is a good idea to save the loading routine for later use.

The next, and final, step is to prepare a BASIC program that uses the shape table.

**The Hi-Res Main Program** There are some special requirements for a BASIC program that uses the shape table you have built. Generally speaking, the main program should begin by:

1. Setting LOMEM up and away from the area to be occupied by the shape table.
2. Specifying the starting address of the shape table.
3. Specifying the special hi-res and shape-table variables.
4. Calling the routine that loads the shape table.
5. Initializing the hi-res system.

Recall that you can set LOMEM from an Integer BASIC program by POKEing the 2-byte version of the desired LOMEM address into addresses 74 and 75. Assuming that you want LOMEM to be at address 3072, the appropriate POKE statements are:

```
POKE 74,0 : POKE 75,12
```

---

You can set LOMEM anywhere you like, as long as it is higher than the starting address of your shape table and leaves room for building up the shape table below it. We will use a LOMEM of 3072 in the remaining examples, and start the shape table at 2048.

After setting LOMEM, set the starting address of the shape table. That address must be entered as a 2-byte number in addresses 808 and 809, with the lower byte going into location 808. So using a table starting address of 2048, the appropriate POKE statements are:

**POKE 808,0 : POKE 809,8**

Addresses 808 and 809 carry the starting address of the shape table. The starting address must be entered as a 2-byte decimal number, with the least-significant byte going into 808.

Again, you have the option of locating the start of the shape table anywhere you wish, just as long as it is lower than your prescribed LOMEM setting and leaves room below LOMEM for the entire table. Thus, virtually every program that uses hi-res shape table techniques ought to begin this way:

---

```
100 POKE 74,0: POKE 75,12
110 POKE 808,0: POKE 809,8
```

---

Recall that a program using hi-res operations must define variables XX, YY, and COLR before doing anything else that uses a variable name. When working with a shape table, that list of critical variables has to be extended to include SHAPE, ROT, and SCALE. You can name them anything else you want as long as your custom names have the same number of characters in them. We will be staying with SHAPE, ROT, and SCALE.

Incidentally, you must define all six of those variables, even if you do not plan to use some of them. So the programming to this point ought to look like this:

---

```
100 POKE 74,0: POKE 75,12
110 POKE 808,0: POKE 809,8
120 XX=YY=COLR=SHAPE=ROT=SCALE
```

---



The SHAPE variable indicates which shape you are dealing with. If your table has four shapes in it, the legitimate values later assigned to the SHAPE variable are 1, 2, 3, and 4.

The ROT variable is used for rotating the designated shape about its starting point. The values assigned to ROT later in the program can be any integer from 0 to 255, but you can see in Table 7-10 that values 0 through 64 are adequate for a full 360 degrees of rotation.

The SCALE variable fixes the size of the designated shape in relation to its size as indicated in the table. The value ultimately assigned to SCALE can be any integer value from 0 to 255. Using a scaling factor of 0 is pointless, because it reduces the size to nothing; attempting to use scaling factors larger than 4 expands the shape to a size that renders it virtually useless. A scaling factor of 1 causes the system to plot the shape in the same point-for-point size you used in designing it originally. You probably realize by now that the SCALE variable cannot properly reduce the size of a figure relative to its original, shape-table specification. It does a nice job of expanding it by factors of 2 or 3, however.

After the program line that defines the six hi-res variables, the next line ought to call the table-loading routine. That is a matter of assigning the full decimal starting address of the shape table to variable ST and doing a GOSUB 1000. *The value assigned to variable ST must be a full decimal version of the 2-byte address POKEd into addresses 808 and 809.* If you forget to use the full decimal version, you will get some discouraging results.

By now, the opening portion of the BASIC program should look like this:

---

```
100 POKE 74,0: POKE 75,12
110 POKE 808,0: POKE 809,8
120 XX=YY=COLR=SHAPE=ROT=SCALE
130 ST=2048: GOSUB 1000
```

---

Now you are free to carry out any operations that aren't directly related to high-resolution graphics. That would include some text and lo-res operations. For our immediate purposes, though, assume that it is time to get into the hi-res operating mode by CALLing INIT at -12288.

Once in the hi-res operating mode, you can fiddle around with the normal PLOT and LINE techniques described earlier in this chapter. But when it is time to draw one of the shapes in your shape table, you must prepare the way by:

1. Making certain that the XX and YY starting position of the shape is clearly defined.

**Table 7-10. ROT Values**

<b>ROT Value</b>	<b>Angle of Rotation (degrees)</b>
0	0.00
2	11.25
4	22.50
6	33.75
8	45.00
10	56.25
12	67.50
14	78.75
16	90.00
18	101.25
20	112.50
22	123.75
24	135.00
26	146.25
28	157.50
30	168.75
32	180.00
34	191.25
36	202.50
38	213.75
40	225.00
42	236.25
44	247.50
46	258.75
48	270.00
50	281.25
52	292.50
54	303.75
56	315.00
58	326.25
60	337.50
62	348.75
64	360.00

2. Assigning a color code for the shape to COLR.
3. Designating which shape you want to draw by assigning the shape number to SHAPE.
4. Designating a ROT angle and SCALE factor.
5. CALLing the shape DRAW routine at address -11456.

All of that is illustrated for you in lines 150 through 190 in Listing 7-6.

Line 150 sets XX and YY to hi-res location 20,30 and calls the POSN routine. Any other operation that leaves XX and YY at some clearly defined position will suffice.

Lines 160 and 170 set the shape color to white and designate shape number 2. Shape 2 in this case is the little creature of Fig. 7-3. Line 180 sets the rotation to 0 (no rotation at all), and then establishes a scale factor of 2. Finally, line 190 calls the shape-drawing routine at address -11465.

DRAW at address -11465 draws the designated shape.

Assuming that you already have the shape-table loading routine (program lines 1000 through 1270) loaded into the system, add the main program at lines 100 through 200, and give it a RUN.

Experiment with the SCALE factor in line 180, try changing the COLR in line 160, and set up some different drawing positions in line 150. Specify shape 1 in line 170 to see the rectangle figure from Fig. 7-2.

Can you think of a way to rewrite the program so that the system will duplicate shape 2 a number of times and at different places on the screen?

Try drawing shape 2 at some position on the screen, and then set up the program for drawing shape 1 by means of the DRAW1 routine at address -11462. Do not specify a position for shape 1, and you will be able to appreciate the purpose of DRAW1. (It begins drawing shape 1 where shape 2 left off.)

CALLing the FIND routine at address -11780 automatically positions the XX and YY variables at the last point specified for a shape just drawn on the screen. After doing FIND, you can specify a PLOT or LINE from that position.

DRAW1 is a shape-linking function. FIND does the same sort of thing, but links a PLOT or LINE to a previously drawn shape.

As you play around with shape tables, you will find that the colors of the shapes often do some unexpected and often undesirable things. The plotting of shapes against various background colors follows the same general limitations described earlier for plotting points and lines. A shape having a white color assigned to it might appear green or blue against a black background, depending on whether the current XX value is even or odd. Portions of a shape will appear white only if there are two points plotted in

## Listing 7-6. Drawing Two Shapes.

---

```
100 POKE 74,0: POKE 75,12
110 POKE 808,0: POKE 809,8
120 XX=YY=COLR=SHAPE=ROT=SCALE
130 ST=2048: GOSUB 1000
140 CALL -12288
150 XX=20:YY=30: CALL -11527
160 COLR=127
170 SHAPE=2
180 ROT=0:SCALE=2
190 CALL -11465
200 END
1000 REM      ** LOAD SHAPE TABLE **
1010 POKE ST+0,2: POKE ST+1,0
1020 POKE ST+2,6: POKE ST+3,0
1030 POKE ST+4,39: POKE ST+5,0
1040 REM      ** SHAPE 1 **
1050 FOR N=6 TO 12: POKE ST+N,45: NEXT N
1060 POKE ST+13,5
1070 FOR N=14 TO 20: POKE ST+N,54: NEXT N
1080 POKE ST+21,6
1090 FOR N=22 TO 28: POKE ST+N,63: NEXT N
1100 POKE ST+29,7
1110 FOR N=30 TO 36: POKE ST+N,36: NEXT N
1120 POKE ST+37,4: POKE ST+38,0
1130 REM      ** SHAPE 2 **
1140 FOR N=39 TO 70: GOSUB 1200+N
1150 POKE ST+N,K: NEXT N
1160 RETURN
1170 REM
1180 REM
1239 K=54: RETURN
1240 K=54: RETURN
1241 K=5: RETURN
1242 K=128: RETURN
1243 K=128: RETURN
1244 K=36: RETURN
1245 K=36: RETURN
1246 K=1: RETURN
1247 K=54: RETURN
1248 K=2: RETURN
1249 K=54: RETURN
1250 K=6: RETURN
1251 K=5: RETURN
1252 K=128: RETURN
1253 K=128: RETURN
1254 K=36: RETURN
1255 K=36: RETURN
1256 K=5: RETURN
1257 K=54: RETURN
1258 K=2: RETURN
1259 K=54: RETURN
1260 K=6: RETURN
1261 K=5: RETURN
1262 K=128: RETURN
1263 K=128: RETURN
1264 K=36: RETURN
1265 K=4: RETURN
1266 K=5: RETURN
1267 K=2: RETURN
1268 K=54: RETURN
1269 K=54: RETURN
1270 K=0: RETURN
```

successive horizontal positions. Also, you will find a big difference between WHITE 127 and WHITE 255. As mentioned earlier, this particular situation is beyond the scope of this book.

**HI-RES VIDEO ADDRESSES** It is possible to build fine hi-res graphics by POKEing values directly to the primary or secondary pages of hi-res video memory. Unfortunately, it is an exceedingly tricky procedure to achieve the color combinations you want. Fortunately, the PLOT, LINE, and DRAW routines already described in this chapter can do the same sort of task equally well. For the sake of completeness, however, Tables 7-11 and 7-12 show the line-by-line memory map for both the primary and secondary pages of hi-res video memory.

**Table 7-11. Hi-Res Primary-Page Memory Map**

Line	Address Range
LINE 0	8192-8231
LINE 1	9216-9255
LINE 2	10240-10279
LINE 3	11264-11303
LINE 4	12288-12327
LINE 5	13312-13351
LINE 6	14336-14375
LINE 7	15360-15399
LINE 8	8320-8359
LINE 9	9344-9383
LINE 10	10368-10407
LINE 11	11392-11431
LINE 12	12416-12455
LINE 13	13440-13479
LINE 14	14464-14503
LINE 15	15488-15527
LINE 16	8448-8487
LINE 17	9472-9511
LINE 18	10496-10535
LINE 19	11520-11559
LINE 20	12544-12583
LINE 21	13568-13607
LINE 22	14592-14631
LINE 23	15616-15655

**Table 7-11—cont. Hi-Res Primary-Page Memory Map**

Line	Address Range
LINE 24	8576–8615
LINE 25	9600–9639
LINE 26	10624–10663
LINE 27	11648–11687
LINE 28	12672–12711
LINE 29	13696–13735
LINE 30	14720–14759
LINE 31	15744–15783
LINE 32	8704–8743
LINE 33	9728–9767
LINE 34	10752–10791
LINE 35	11776–11815
LINE 36	12800–12839
LINE 37	13824–13863
LINE 38	14848–14887
LINE 39	15872–15911
LINE 40	8832–8871
LINE 41	9856–9895
LINE 42	10880–10919
LINE 43	11904–11943
LINE 44	12928–12967
LINE 45	13952–13991
LINE 46	14976–15015
LINE 47	16000–16039
LINE 48	8960–8999
LINE 49	9984–10023
LINE 50	11008–11047
LINE 51	12032–12071
LINE 52	13056–13095
LINE 53	14080–14119
LINE 54	15104–15143
LINE 55	16128–16167
LINE 56	9088–9127
LINE 57	10112–10151
LINE 58	11136–11175
LINE 59	12160–12199
LINE 60	13184–13223

**Table 7-11—cont. Hi-Res Primary-Page Memory Map**

Line	Address Range
LINE 61	14208–14247
LINE 62	15232–15271
LINE 63	16256–16295
LINE 64	8232–8271
LINE 65	9256–9295
LINE 66	10280–10319
LINE 67	11304–11343
LINE 68	12328–12367
LINE 69	13352–13391
LINE 70	14376–14415
LINE 71	15400–15439
LINE 72	8360–8399
LINE 73	9384–9423
LINE 74	10408–10447
LINE 75	11432–11471
LINE 76	12456–12495
LINE 77	13480–13519
LINE 78	14504–14543
LINE 79	15528–15567
LINE 80	8488–8527
LINE 81	9512–9551
LINE 82	10536–10575
LINE 83	11560–11599
LINE 84	12584–12623
LINE 85	13608–13647
LINE 86	14632–14671
LINE 87	15656–15695
LINE 88	8616–8655
LINE 89	9640–9679
LINE 90	10664–10703
LINE 91	11688–11727
LINE 92	12712–12751
LINE 93	13736–13775
LINE 94	14760–14799
LINE 95	15784–15823
LINE 96	8744–8783

**Table 7-11—cont. Hi-Res Primary-Page Memory Map**

Line	Address Range
LINE 97	9768–9807
LINE 98	10792–10831
LINE 99	11816–11855
LINE 100	12840–12879
LINE 101	13864–13903
LINE 102	14888–14927
LINE 103	15912–15951
LINE 104	8872–8911
LINE 105	9896–9935
LINE 106	10920–10959
LINE 107	11944–11983
LINE 108	12968–13007
LINE 109	13992–14031
LINE 110	15016–15055
LINE 111	16040–16079
LINE 112	9000–9039
LINE 113	10024–10063
LINE 114	11048–11087
LINE 115	12072–12111
LINE 116	13096–13135
LINE 117	14120–14159
LINE 118	15144–15183
LINE 119	16168–16207
LINE 120	9128–9167
LINE 121	10152–10191
LINE 122	11176–11215
LINE 123	12200–12239
LINE 124	13224–13263
LINE 125	14248–14287
LINE 126	15272–15311
LINE 127	16296–16335
LINE 128	8272–8311
LINE 129	9296–9335
LINE 130	10320–10359
LINE 131	11344–11383
LINE 132	12368–12407
LINE 133	13392–13431
LINE 134	14416–14455



**Table 7-11—cont. Hi-Res Primary-Page Memory Map**

<b>Line</b>	<b>Address Range</b>
LINE 135	15440–15479
LINE 136	8400–8439
LINE 137	9424–9463
LINE 138	10448–10487
LINE 139	11472–11511
LINE 140	12496–12535
LINE 141	13520–13559
LINE 142	14544–14583
LINE 143	15568–15607
LINE 144	8528–8567
LINE 145	9552–9591
LINE 146	10576–10615
LINE 147	11600–11639
LINE 148	12624–12663
LINE 149	13648–13687
LINE 150	14672–14711
LINE 151	15696–15735
LINE 152	8656–8695
LINE 153	9680–9719
LINE 154	10704–10743
LINE 155	11728–11767
LINE 156	12752–12791
LINE 157	13776–13815
LINE 158	14800–14839
LINE 159	15824–15863
LINE 160	8784–8823
LINE 161	9808–9847
LINE 162	10832–10871
LINE 163	11856–11895
LINE 164	12880–12919
LINE 165	13904–13943
LINE 166	14928–14967
LINE 167	15952–15991
LINE 168	8912–8951
LINE 169	9936–9975
LINE 170	10960–10999

**Table 7-11—cont. Hi-Res Primary-Page Memory Map**

Line	Address Range
LINE 171	11984–12023
LINE 172	13008–13047
LINE 173	14032–14071
LINE 174	15056–15095
LINE 175	16080–16119
LINE 176	9040–9079
LINE 177	10064–10103
LINE 178	11088–11127
LINE 179	12112–12151
LINE 180	13136–13175
LINE 181	14160–14199
LINE 182	15184–15223
LINE 183	16208–16247
LINE 184	9168–9207
LINE 185	10192–10231
LINE 186	11216–11255
LINE 187	12240–12279
LINE 188	13264–13303
LINE 189	14288–14327
LINE 190	15312–15351
LINE 191	16336–16375

**Table 7-12. Hi-Res Secondary-Page Memory Map**

Line	Address Range
LINE 0	16384–16423
LINE 1	17408–17447
LINE 2	18432–18471
LINE 3	19456–19495
LINE 4	20480–20519
LINE 5	21504–21543
LINE 6	22528–22567
LINE 7	23552–23591
LINE 8	16512–16551
LINE 9	17536–17575
LINE 10	18560–18599
LINE 11	19584–19623
LINE 12	20608–20647

**Table 7-12—cont. Hi-Res Secondary-Page Memory Map**

Line	Address Range
LINE 13	21632–21671
LINE 14	22656–22695
LINE 15	23680–23719
LINE 16	16640–16679
LINE 17	17664–17703
LINE 18	18688–18727
LINE 19	19712–19751
LINE 20	20736–20775
LINE 21	21760–21799
LINE 22	22784–22823
LINE 23	23808–23847
LINE 24	16768–16807
LINE 25	17792–17831
LINE 26	18816–18855
LINE 27	19840–19879
LINE 28	20864–20903
LINE 29	21888–21927
LINE 30	22912–22951
LINE 31	23936–23975
LINE 32	16896–16935
LINE 33	17920–17959
LINE 34	18944–18983
LINE 35	19968–20007
LINE 36	20992–21031
LINE 37	22016–22055
LINE 38	23040–23079
LINE 39	24064–24103
LINE 40	17024–17063
LINE 41	18048–18087
LINE 42	19072–19111
LINE 43	20096–20135
LINE 44	21120–21159
LINE 45	22144–22183
LINE 46	23168–23207
LINE 47	24192–24231
LINE 48	17152–17191

**Table 7-12—cont. Hi-Res Secondary-Page Memory Map**

Line	Address Range
LINE 49	18176–18215
LINE 50	19200–19239
LINE 51	20224–20263
LINE 52	21248–21287
LINE 53	22272–22311
LINE 54	23296–23335
LINE 55	24320–24359
LINE 56	17280–17319
LINE 57	18304–18343
LINE 58	19328–19367
LINE 59	20352–20391
LINE 60	21376–21415
LINE 61	22400–22439
LINE 62	23424–23463
LINE 63	24448–24487
LINE 64	16424–16463
LINE 65	17448–17487
LINE 66	18472–18511
LINE 67	19496–19535
LINE 68	20520–20559
LINE 69	21544–21583
LINE 70	22568–22607
LINE 71	23592–23631
LINE 72	16552–16591
LINE 73	17576–17615
LINE 74	18600–18639
LINE 75	19624–19663
LINE 76	20648–20687
LINE 77	21672–21711
LINE 78	22696–22735
LINE 79	23720–23759
LINE 80	16680–16719
LINE 81	17704–17743
LINE 82	18728–18767
LINE 83	19752–19791
LINE 84	20776–20815
LINE 85	21800–21839

**Table 7-12—cont. Hi-Res Secondary-Page Memory Map**

Line	Address Range
LINE 86	22824–22863
LINE 87	23848–23887
LINE 88	16808–16847
LINE 89	17832–17871
LINE 90	18856–18895
LINE 91	19880–19919
LINE 92	20904–20943
LINE 93	21928–21967
LINE 94	22952–22991
LINE 95	23976–24015
LINE 96	16936–16975
LINE 97	17960–17999
LINE 98	18984–19023
LINE 99	20008–20047
LINE 100	21032–21071
LINE 101	22056–22095
LINE 102	23080–23119
LINE 103	24104–24143
LINE 104	17064–17103
LINE 105	18088–18127
LINE 106	19112–19151
LINE 107	20136–20175
LINE 108	21160–21199
LINE 109	22184–22223
LINE 110	23208–23247
LINE 111	24232–24271
LINE 112	17192–17231
LINE 113	18216–18255
LINE 114	19240–19279
LINE 115	20264–20303
LINE 116	21288–21327
LINE 117	22312–22351
LINE 118	23336–23375
LINE 119	24360–24399
LINE 120	17320–17359
LINE 121	18344–18383

**Table 7-12—cont. Hi-Res Secondary-Page Memory Map**

Line	Address Range
LINE 122	19368–19407
LINE 123	20392–20431
LINE 124	21416–21455
LINE 125	22440–22479
LINE 126	23464–23503
LINE 127	24488–24527
LINE 128	16464–16503
LINE 129	17488–17527
LINE 130	18512–18551
LINE 131	19536–19575
LINE 132	20560–20599
LINE 133	21584–21623
LINE 134	22608–22647
LINE 135	23632–23671
LINE 136	16592–16631
LINE 137	17616–17655
LINE 138	18640–18679
LINE 139	19664–19703
LINE 140	20688–20727
LINE 141	21712–21751
LINE 142	22736–22775
LINE 143	23760–23799
LINE 144	16720–16759
LINE 145	17744–17783
LINE 146	18768–18807
LINE 147	19792–19831
LINE 148	20816–20855
LINE 149	21840–21879
LINE 150	22864–22903
LINE 151	23888–23927
LINE 152	16848–16887
LINE 153	17872–17911
LINE 154	18896–18935
LINE 155	19920–19959
LINE 156	20944–20983
LINE 157	21968–22007
LINE 158	22992–23031

**Table 7-12—cont. Hi-Res Secondary-Page Memory Map**

Line	Address Range
LINE 159	24016–24055
LINE 160	16976–17015
LINE 161	18000–18039
LINE 162	19024–19063
LINE 163	20048–20087
LINE 164	21072–21111
LINE 165	22096–22135
LINE 166	23120–23159
LINE 167	24144–24183
LINE 168	17104–17143
LINE 169	18128–18167
LINE 170	19152–19191
LINE 171	20176–20215
LINE 172	21200–21239
LINE 173	22224–22263
LINE 174	23248–23287
LINE 175	24272–24311
LINE 176	17232–17271
LINE 177	18256–18295
LINE 178	19280–19319
LINE 179	20304–20343
LINE 180	21328–21367
LINE 181	22352–22391
LINE 182	23376–23415
LINE 183	24400–24439
LINE 184	17360–17399
LINE 185	18384–18423
LINE 186	19408–19447
LINE 187	20432–20471
LINE 188	21456–21495
LINE 189	22480–22519
LINE 190	23504–23543
LINE 191	24528–24567





# Using Short Machine-Language Routines With BASIC

## 8

Anyone who has done even a modest amount of programming in Apple Integer BASIC ought to be familiar with some common CALL statements. Perhaps the most-used of these is CALL -936. That statement calls a subroutine within the Apple monitor that homes the cursor and clears the text screen.

If you have been following the previous discussions, you are also familiar with a whole family of useful CALL statements. Most of them deal with setting the cursor position on the screen or clearing parts of the screen. In all of those instances, the CALL statements refer to the starting address of a monitor subroutine that is written in machine language. Those machine-language subroutines all end with a code that returns the system to the BASIC program that called the subroutine.

Fortunately for programmers who know nothing at all about machine-language programming, those commonly used CALL statements refer to machine-language subroutines that are complete in themselves. Anyone can use them without giving the slightest thought to machine-language techniques.

The Apple monitor, however, is a vast repository of machine-language routines that can be executed by doing a little bit of machine coding. Such routines often run much faster than their BASIC counterparts. Once you understand some of the fundamentals of machine-language programming, you will be able to use these routines also. But they include a few lines of POKE statements that deposit short machine-language subroutines into memory.

**A FEW USEFUL MACHINE INSTRUCTIONS**      Most of the monitor subroutines use the A, X, and Y registers in the microprocessor. These

three registers may be thought of as special memory locations that can hold decimal values from 0 to 255—the same range of values that can be POKEd or PEEKed in any other memory location. These registers are different, however, in that we cannot address them directly from BASIC. Instead, we must use machine language to address them.

**Loading Data to the Registers** You can load a number between 0 and 255 into the A, X, and Y registers. The idea is quite similar to POKing data to a RAM address in memory, but as mentioned earlier, the procedure is a bit different.

There are three machine-language instructions for loading data to the registers. They are 169, 162, and 160, and they load data into registers A, X, and Y, respectively.

169 *data* loads *data* directly into the A register.  
162 *data* loads *data* directly into the X register.  
160 *data* loads *data* directly into the Y register.

In all three instructions, *data* is an integer between 0 and 255 that you want to load into the designated register. So if you want to load a 27 into the A register, you would write a program that deals with two numbers in succession: 169 and 27. The first number, 169, tells the microprocessor to accept data directly into its A register. The second number, 27, is the data to be accepted by the A register.

On the other hand, if you want to load that number 27 to the X register, the appropriate sequence of numbers is 162 followed by 27. And if you want to load the number to the Y register, the number sequence is 160 27.

You can also load the A, X, and Y registers with data from any address in memory. The coding is a bit trickier than that required for loading just a number, because you must break down the memory address into two decimal parts.

The problem, you see, is that machine language uses sequences of code numbers having values limited to the range of 0 through 255, and most RAM addresses have numbers much larger than that.

There are three machine-language instructions for loading data from memory into the registers. They are 173, 174, and 172, and they load RAM data into registers A, X, and Y, respectively.

Each of those instructions requires a sequence of three machine codes. The first code is the instruction. It designates the nature of the operation and points to a particular register. For instance, 173 designates a loading operation from RAM into the A register. The two codes following the instruction represent the address of the data. The least-significant, or low portion, comes first, followed by the most-significant, or high portion.

173 *addrL addrH* loads the content of a memory address represented as *addrL* and *addrH* into the A register.

174 *addrL addrH* loads the content of a memory address represented as *addrL* and *addrH* into the X register.

172 *addrL addrH* loads the content of a memory address represented as *addrL* and *addrH* into the Y register.

What three codes are necessary for loading the A register with the data in RAM address 825? Well, the first code in the sequence is 173. The two following code numbers represent address 825, broken down into the low and high decimal parts. In this case, *addrL*=57, and *addrH*=3. So the proper code sequence is:

173 57 3

What would the following code sequence do?

172 44 2

The first number in the series indicates a loading operation to the Y register from some memory address. The address, here coded as 44 2, is actually address 554.

See if you can work out the coding sequence for loading the X register with data contained in RAM address 1024.

Table 8-1 summarizes the six register-loading instructions cited thus far. The list also includes the standard *mnemonics* (pronounced *nee-MON-ics*) for each coding sequence. The mnemonics describe the operations in a shorthand form that is far more meaningful to programmers than the actual machine-language sequences are.

Suppose you are preparing a program that will load a value of 55 di-

**Table 8-1. Register-Loading Instructions**

Mnemonic	Machine Code	Definition
LDA <i>#data</i>	169 <i>data</i>	Loads <i>data</i> immediately to register A
LDA <i>addr</i>	173 <i>addrL addrH</i>	Loads the content of <i>addr</i> to register A
LDX <i>#data</i>	162 <i>data</i>	Loads <i>data</i> immediately to register X
LDX <i>addr</i>	174 <i>addrL addrH</i>	Loads the content of <i>addr</i> to register X
LDY <i>#data</i>	160 <i>data</i>	Loads <i>data</i> immediately to register Y
LDY <i>addr</i>	172 <i>addrL addrH</i>	Loads the content of <i>addr</i> to register Y

rectly to the A register. As you saw earlier in this discussion, the proper machine-language sequence would be:

169 55

The microprocessor can understand the coding sequence, but programmers find it difficult. A more human-oriented way to express the same operation is by writing this:

LDA #55

Literally interpreted, that *assembly-language* instruction says: Load a value of 55 into the A register.

What, then, is the literal interpretation of the following assembly-language instruction?

LDY 1020

It means: Load the Y register with data contained in address 1020. The machine-language sequence would be:

172 252 3

**Storing Data from the Registers** Just as there are machine-language instructions for loading data into the registers, there are machine-language instructions for storing the contents of the registers in memory. There are just three such instructions used in this chapter. They are 141, 142, and 140, and they store data from registers A, X, and Y, respectively.

141 *addrL addrH* stores the content of the A register at an address represented by *addrL* and *addrH*.

142 *addrL addrH* stores the content of the X register at an address represented by *addrL* and *addrH*.

140 *addrL addrH* stores the content of the Y register at an address represented by *addrL* and *addrH*.

The instructions are all three-code sequences. The first code designates the operation (store) and the register involved (A, X, or Y). The two final codes indicate the address that is to accept the data.

Suppose it is necessary to store the content of the A register at RAM address 800. The appropriate machine-language sequence for doing that is:

141 32 3

where 141 is the instruction, 32 is *addrL* for address 800, and 3 is *addrH* for address 800.

Table 8-2 summarizes these three instructions along with their mnemonics.

The assembly-language version of the example just cited is

STA 800

It means: Store the content of register A at address 800.

Table 8-2. Register-Reading Instructions

Mnemonic	Machine Code	Definition
STA <i>addr</i>	141 <i>addrL addrH</i>	Stores the content of register A at <i>addr</i>
STX <i>addr</i>	142 <i>addrL addrH</i>	Stores the content of register X at <i>addr</i>
STY <i>addr</i>	140 <i>addrL addrH</i>	Stores the content of register Y at <i>addr</i>

**Going and Returning** There are two more helpful machine-language instructions. One tells the microprocessor where to begin executing a machine-language subroutine, and the other tells the system to return to the main routine. They are much like the BASIC GOSUB and RETURN statements. Table 8-3 shows those instructions and their assembly-language, mnemonic forms.

The assembly-language instruction JSR *addr* tells the microprocessor to jump to a routine that begins at address *addr*, and keep track of where to return when the routine is done. JSR 841, for example, means: Jump to a routine beginning at address 841, and keep track of the place to return. The corresponding machine-language version of that instruction is:

32 73 3

The first code, 32, is the instruction, 73 is the *addrL* part of address 841, and 3 is the *addrH* part.

Table 8-3. Jump-to-Subroutine and Return Instructions

Mnemonic	Machine Code	Definition
JSR <i>addr</i>	32 <i>addrL addrH</i>	Executes a subroutine that begins at <i>addr</i>
RTS	96	Returns operations to the calling routine

The RTS assembly-language instruction is a one-code instruction—96. Whenever the microprocessor encounters the number 96 as an instruction, it returns to the place in a program that is previously saved by a JSR instruction. Most machine-language routines end with an RTS instruction.

**Some Preliminary Examples** Before seeing exactly how machine language ought to be presented to the microprocessor, consider a few examples of the kind of thinking that goes into composing such programs.

*Example 1:* Write a machine-language sequence that stores character code 20 at address 1024. Recall that 20 is the character code for an inverse space and that 1024 is the video RAM address of the first point in the upper left-hand corner of the screen. The routine, then, should plot a square of light at that point on the screen.

The program sequence goes like this:

1. Load a value of 20 directly into the A register.
2. Store that value at address 1024.

The assembly-language version looks like this:

```
LDA #20    ;LOAD INVERSE SPACE TO REGISTER A
STA 1024   ;STORE IT TO VIDEO ADDRESS 1024
```

The explanations following each instruction and separated from the instruction by a semicolon are called *comments*. Their only purpose, if they are used at all, is to help the programmer remember the purpose of the instruction. Comments in assembly-language programs are like REMs in BASIC.

The microprocessor cannot understand the program as presented in this assembly-language form. So one step remains—to turn the instructions into their machine-language form. Referring to the tables shown earlier, those two machine-language instructions look like this:

```
169 20
141 0 4
```

Combining both the assembly-language version (often called the *source-code* version) and the machine-language version (*object-code* version) into a single presentation, we have:

```
169 20    LDA #20 ;LOAD INVERSE SPACE TO REGISTER A
141  0 4  STA 1024 ;STORE IT TO VIDEO ADDRESS 1024
```

The left-hand column, containing the machine-language version of the program, is the *object-code field*. The middle column, containing the assembly-language version of the program, is the *source-code field*. The last column, containing the comments, is the *comment field*.

*Example 2:* Fetch the character code from video address 1024 and load it to video address 1025. In effect, we want to shift the character in the first space in the upper left-hand corner of the screen one place to the right.

The programming sequence ought to go something like this:

1. Load the contents of address 1024 into the A register in the micro-processor.
2. Load the contents of the A register into address 1025 in the video memory.

The assembly-language version of those two steps is:

```
LDA 1024    ;FETCH THE CODE FROM ADDRESS 1024
STA 1025    ;STORE THE CODE TO ADDRESS 1025
```

The machine-language version is:

```
173 0 4
141 1 4
```

Together, they look like this:

```
173 0 4    LDA 1024    ;FETCH THE CODE FROM ADDRESS 1024
141 1 4    STA 1024    ;STORE THE CODE TO ADDRESS 1025
```

*Example 3:* Set up a machine program that

1. Loads a value of 16 directly into the X register.
2. Loads a value of 44 directly into the A register.
3. Calls a machine-language subroutine that begins at address 845.
4. Returns to the calling routine.

We begin with the assembly-language version:

```
LDX #16      ;LOAD 16 TO THE X REGISTER
LDA #44      ;LOAD 44 TO THE A REGISTER
JSR 845      ;CALL THE ROUTINE AT ADDRESS 845
RTS          ;RETURN TO THE CALLING ROUTINE
```

Then, we generate the machine-language sequence:

```
162 16
169 44
32 77 3
96
```

Next, we put it all together:

```
162 16    LDS #16    ;LOAD 16 TO THE X REGISTER
169 44    LDA #44    ;LOAD 44 TO THE A REGISTER
32 77 3    JSR 845    ;CALL THE ROUTINE AT ADDRESS 845
96         RTS       ;RETURN TO THE CALLING ROUTINE
```

## ENTERING AND RUNNING MACHINE-LANGUAGE ROUTINES

The microprocessor reads and executes machine language directly from memory. The microprocessor expects to find the codes residing in a strict sequential order and executes them that way. There are no line numbers; rather, the codes reside in a block of memory. The memory addresses, and not line numbers, are the only real organizers for machine-language programs.

So before you begin entering machine-language programs, you must determine where they will be deposited in RAM. You certainly don't want to put them into video RAM, because that memory is dedicated to another application. And you shouldn't try putting them into RAM that might be used by BASIC programs. In short, you must use RAM that will not be used for anything else.

There happens to be some space in RAM that isn't used at all if you are not running under DOS or using certain lineprinters. Even if you are using DOS, this RAM space is used only while booting up the system, so you can use the space as long as you don't boot up DOS after loading your machine codes. This largely unused block of RAM extends from address 768 to 1023; it is just below the section devoted to low-resolution graphics and text. That is a great place to deposit short machine-language routines and any variables such routines might use.

Fig. 8-1 is a memory map that we will be using for most machine-language routines. It sets aside addresses 800 through 899 for short machine-language routines. There is enough space for 100 individual codes in that range. The upper part of that memory map can be used for saving variables that are required for executing the routines. Such a block of memory, here shown between addresses 900 and 924, is called a *scratch-pad* memory.



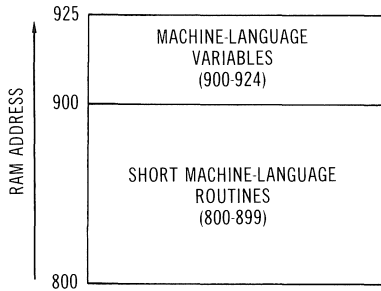


Fig. 8-1. Machine-language memory map.

In principle, you are free to use other RAM space anywhere else in the usable part of memory, but I have chosen this block of memory because it is never disturbed by BASIC programs.

When it is time to enter the machine-language routine, POKE the codes into successively higher addresses in your block of memory. So if a particular routine begins with a code sequence such as:

```
173 0 4
```

enter it this way:

```
POKE 800,173
POKE 801,0
POKE 802,4
```

And if you want to continue the program with another sequence such as:

```
141 1 4
```

continue the entry process with:

```
POKE 803,141
POKE 804,1
POKE 805,4
```

The example thus far shows the two sequences of three codes occupying RAM addresses 800 through 805. If there were more to the program, you would continue POKEing codes into successively higher RAM addresses until they were all stored.

Consider this complete routine:

```
169 30      LDA #30      ;LOAD 30 TO REGISTER A
141 44 0    STA 44        ;STORE IT IN ADDRESS 44
169 15      LDA #15      ;LOAD 15 TO REGISTER A
160 12      LDY #12      ;LOAD 12 TO REGISTER Y
32 25 248   JSR -2023     ;CALL A ROUTINE AT -2023
96          RTS          ;RETURN TO CALLING ROUTINE
                        ;IN BASIC
```

The machine-language instructions in that routine are entered into the system as follows:

```
POKE 800,169
POKE 801,30
POKE 802,141
POKE 803,44
POKE 804,0
POKE 805,169
POKE 806,15
POKE 807,160
POKE 808,12
POKE 809,32
POKE 810,25
POKE 811,248
POKE 812,96
```

That machine-language program now occupies RAM locations 800 through 812, and you can CALL 800 from BASIC to execute it.

Here is a complete rendition of the program, including the addresses of the first code in each instruction:

```
800 169 30      LDA #30      ;LOAD 30 TO REGISTER A
802 141 44 0    STA 44        ;STORE IT IN ADDRESS 44
805 169 15      LDA #15      ;LOAD 15 TO REGISTER A
807 160 12      LDY #12      ;LOAD 12 TO REGISTER Y
809 32 25 248   JSR -2023     ;CALL A ROUTINE AT -2023
812 96          RTS          ;RETURN TO CALLING ROUTINE
                        ;IN BASIC
```

That says everything that needs to be said about the program. In left-to-right order, it shows the address of the first code in each instruction, the object-code version of each instruction, the source-code version of each instruction, and an appropriate comment.

To check out the program, POKE the series of 13 codes into addresses 800 through 812. Then enter and run this BASIC routine:

---

```
10 CALL -936
20 GOSUB 100
30 GR
40 COLOR=9
50 CALL 800
60 END
```

---

If you have POKEd the machine-language routine into addresses 800 to 812, as suggested, you should see an orange line drawn on the screen. Notice that the BASIC routine simply sets up the graphics mode and color code 9. Line 40 CALLs your machine-language routine—a machine-language version of the BASIC HLIN statement. After executing the machine-language program, the system returns to BASIC line 50 and comes to an END.

You can now run that BASIC program any number of times, and each time it will draw the line by calling up your machine-language routine. In fact, you can do a NEW to wipe out the BASIC portion of the program without disturbing the language routine. Try it, then re-enter the BASIC program and run it. Indeed, the machine-language routine is not affected by BASIC operations. Of course, turning off the computer erases the routine. But you can save it by including the POKEing operation in the BASIC program that uses it. Consider this:

---

```
10 CALL -936
20 GOSUB 100
30 GR
40 COLOR=9
50 CALL 800
60 END
100 POKE 800,169: POKE 801,30
110 POKE 802,141: POKE 803,44: POKE 804,0
120 POKE 805,169: POKE 806,15
130 POKE 807,160: POKE 808,12
140 POKE 809,32: POKE 810,25: POKE 811,248
150 POKE 812,96
160 RETURN
```

---

That BASIC program POKEs the machine-language routine by means of a subroutine in lines 100 through 160. After that, it executes the operations for setting up the graphics mode and calling the machine-language routine from program line 50. BASIC programs that call custom machine-

---

language programs are generally written in this fashion. These BASIC programs include the POKE subroutine (often called the *loader* routine) necessary for loading the machine language into memory. They also include the BASIC routines that call the machine-language routine at the moment. The entire program can be saved on cassette or disk without having to worry about loading or saving the machine-language routine separately.

**CALLING SOME MONITOR ROUTINES** Recall from discussions in previous chapters that there is a family of monitor routines that can be called directly from BASIC. Those routines, summarized in Table 8-4, require no special setup with machine-language instructions. However, there is no reason machine language may not be used. For example, the normal way to execute a HOME operation is by doing a CALL -936 from BASIC. You can do the same thing with machine language by executing the sequence 32 88 252. That sequence represents a JSR -936. The other instructions in the table also begin with a code 32, or JSR, instruction.

The main purpose of this discussion, though, is to introduce some other monitor routines that cannot be called directly from BASIC, or to be more precise, that require some machine-language routines.

**Printing With STOADV** The STOADV routine prints a single character on the screen and advances the cursor position. The routine begins at address -1040, but it cannot be properly executed unless you first load the

**Table 8-4. Monitor Routines Available to BASIC**

<b>Routine</b>	<b>From BASIC</b>	<b>From Machine Language</b>
Linefeed/Carriage Return	CALL -926	32 98 252
Home and clear	CALL -936	32 88 252
Advance the cursor	CALL -1036	32 244 251
Backspace the cursor	CALL -1008	32 16 252
Upward linefeed	CALL -998	32 26 252
Downward linefeed	CALL -922	32 102 252
Clear to end of line	CALL -868	32 156 252
Clear to end of page	CALL -958	32 68 252
Clear top of mixed graphics	CALL -1994	32 54 248
Clear full-screen graphics	CALL -1998	32 50 248
Wait for any keystroke	CALL -741	32 27 253

desired character code into the A register. Thus, an appropriate assembly-language routine using STOADV looks like this:

```
LDA #CHAR      ;LOAD DESIRED CHARACTER CODE TO A
JSR -1040       ;CALL STOADV TO PRINT IT
RTS             ;RETURN TO BASIC
```

The operator CHAR represents the character code of the character to be printed on the screen.

If the character to be printed happens to be a flashing X, the complete machine program takes this form:

```
800 169 88      LDA #88      ;LOAD FLASHING X TO REGISTER A
802 32 240 251  JSR -1040     ;CALL STOADV TO PRINT IT
805 96          RTS          ;RETURN TO BASIC
```

That would specify the setup routine in RAM locations 800 through 805. An appropriate BASIC routine for doing the actual programming could look like this:

---

```
100 POKE 800,169: POKE 801,88
110 POKE 802,32: POKE 803,240: POKE 804,251
120 POKE 805,96
```

---

Enter and run that basic program, and you will have the machine-language routine loaded into the system. You won't see anything happening as the system executes that BASIC program—it simply loads the machine-language routine for you. To see the effect of STOADV, you must now write a BASIC program that calls your machine-language routine. Try this:

---

```
10 CALL -936
20 FOR N=0 TO 255
30 CALL 800
40 NEXT N
50 END
```

---

Assuming that you have previously run the machine-language loader, running this program clears the screen, homes the cursor, and calls the machine-language subroutine 256 times to print the flashing-X characters that many times on the screen.

If you want to change the number of flashing-X characters printed on

the screen, simply revise the BASIC program at line 20. But if you want to change the character that the machine-language subroutine prints, you must POKE a different character code into address 801. For instance, enter POKE 801,32 and run the BASIC calling routine again. Notice this time that the machine-language subroutine loads an inverse-space character into register A.

**Printing With COUT1** COUT1 prints a character to the screen and advances the cursor in much the same way that STOADV does. The only real difference is that COUT1 cannot print any of the “control” character codes between 128 and 159.

The COUT1 monitor routine begins at address -528, and the character to be printed must be residing in register A just prior to the execution of that routine. The assembly-language form of a routine that uses COUT1 looks something like this:

```
LDA #CHAR
JSR -528
RTS
```

The machine-language and assembly-language versions, beginning from RAM address 800, look like this:

```
800 169 24      LDA #24    ;LOAD AN INVERSE X TO REGISTER A
802 32 240 253 JSR -528    ;RUN THE COUT1 ROUTINE
805 96          RTS       ;RETURN TO BASIC
```

That calls for loading character-code 24 into the A register, then printing it to the screen via the COUT1 monitor routine. The custom routine ends with RTS to return control to the BASIC program that called it in the first place.

Given the addresses and machine codes for this printing routine, enter and run this BASIC loader for it:

---

```
100 POKE 800,169: POKE 801,24
110 POKE 802,32: POKE 803,240: POKE 804,253
120 POKE 805,96
```

---

Once you have run that loading routine, you can delete it without disturbing the machine-language programming.

Try out the custom machine-language routine with a BASIC program such as this one:

---

```
10 CALL -936
20 FOR N=0 TO 255
30 CALL 800
40 NEXT N
50 END
```

---

That calls the custom machine-language routine 256 times in succession, printing an inverse-X character and advancing the cursor each time.

**Printing Blanks With PRBL2** A monitor routine called PRBL2 lets you print between 1 and 256 blanks in succession on the screen. Before it can run properly, however, you must load the number of blanks to be printed into the X register. The routine cannot print 0 blanks, so if you load a 0 into the X register, PRBL2 will print 256 blanks in succession.

The general assembly-language form of a routine using PRBL2 looks like this:

```
LDS #BLNK ;LOAD NUMBER OF BLANKS TO REGISTER X
JSR -1718 ;CALL THE PRBL2 MONITOR ROUTINE
RTS      ;RETURN TO BASIC
```

The second instruction indicates that the space-printing routine in the monitor begins at address -1718.

Assuming you want to begin such a routine at address 800, the assembled version takes this form:

```
800 162 40      LDX #40      ;LOAD 40 TO REGISTER X
802 32 74 249   JSR -1718    ;PRINT THE BLANKS
805 96          RTS          ;RETURN TO BASIC
```

The idea is to have the routine print 40 blanks in succession. The BASIC version of the machine-language loader can look like this:

---

```
100 POKE 800,162: POKE 801,40
110 POKE 802,32: POKE 803,74: POKE 804,249
120 POKE 805,96
```

---

After entering and running that machine-language loader, you can print 40 blanks in succession by doing a CALL 800 from BASIC. It is possible to change the number of blanks by POKEing some number other than 40 into address location 801.

**Setting the Low-Resolution Color With SETCOL**     A monitor routine that begins at address -1948 can set up the color to be plotted under low-resolution graphics, provided that you load the color code into the A register first. The routine is called SETCOL, and it is normally executed before doing a different routine that does the actual low-resolution plotting.

Soon you will see how to set up some low-resolution plotting routines from machine language. For now, however, it is more important to see how SETCOL works. The general form is:

```
LDA #COL    ;LOAD THE COLOR CODE (0- 15) TO REGISTER A
JSR -1948   ;EXECUTE SETCOL
RTS         ;RETURN TO BASIC
```

If we choose to load that program from address 800, then the source code and object code look like this:

```
800 169 9          LDA #9          ;LOAD ORANGE TO REGISTER A
802 32 100 248     JSR -1948       ;EXECUTE SETCOL
805 96            RTS             ;RETURN TO BASIC
```

An appropriate loader is:

---

```
100 POKE 800,169: POKE 801,9
110 POKE 802,32: POKE 803,100: POKE 804,248
120 POKE 805,96
```

---

Enter and run that loader to get the custom machine-language routine into memory. After that, test its operation with a BASIC program such as this one:

```
10 GR
20 CALL 800
30 HLIN 0,39 AT 10
40 END
```

The CALL 800 in this case replaces the usual COLOR statement in BASIC. The color set by the machine-language routine is orange, so this program plots an orange bar across the screen. To change the color, simply POKE a different color code right into the custom machine-language program at address 801.

**Using the Monitor Version of PLOT**     The PLOT statement in BASIC is easy to use, but there is a machine-language version of it tucked



away at address -2048 in the monitor. Before using it, you must set up some registers. Specifically, you must load the vertical coordinate into register A, and the horizontal coordinate into register Y.

An assembly-language version of a routine that uses PLOT takes this form:

```
LDA #VERT      ;LOAD VERTICAL COORDINATE TO REGISTER A
LDY #HORZ      ;LOAD HORIZONTAL COORDINATE TO Y
JSR -2048      ;CALL MONITOR'S PLOT
RTS            ;RETURN TO BASIC
```

The following routine begins at address 800 and plots a low-resolution block at coordinates 10 and 8:

```
800 169 8      LDA #8      ;LOAD VERTICAL 8 TO REGISTER A
802 160 10     LDY #10     ;LOAD HORIZ 10 TO REGISTER Y
804 32 0 248   JSR -2048   ;CALL MONITOR'S PLOT ROUTINE
807 96         RTS        ;RETURN TO BASIC
```

Its loader looks like:

---

```
100 POKE 800,169: POKE 801,8
110 POKE 802,160: POKE 803,10
120 POKE 804,32: POKE 805,0: POKE 806,248
130 POKE 807,96
```

---

After entering and running that BASIC loader, test the customized PLOT routine with this BASIC program:

---

```
10 GR
20 COLOR=2
30 CALL 800
40 END
```

---

The CALL 800 in this instance replaces the usual PLOT statement of BASIC. The custom machine-language routine plots COLOR at coordinates 10 and 8. You can alter the coordinates specified by the machine-language routine by POKEing different values (from 0 to 39) into addresses 801 and 803.

You can also replace the color statement in line 20 of the BASIC program with a CALL to a second machine-language routine that sets the plotting color. Or, you might include the SETCOL routine—described in the previous section—the same custom machine-language routine that PLOT is a part of. Consider this assembly-language program:

800	169	9	LDA #9	;ORANGE COLOR CODE TO REGISTER A
802	32	100	248 JSR -1948	;EXECUTE SETCOL
805	169	12	LDA #12	;SET VERTICAL PLOT TO REGISTER A
807	160	10	LDY #10	;SET HORIZ PLOT TO REGISTER Y
809	32	0	248 JSR -2048	;EXECUTE MONITOR'S PLOT
812	96		RTS	;RETURN TO BASIC

This custom machine-language program sets up color code 9, and plots that color at coordinates 10 and 12. The appropriate BASIC loader routine can look like this:

---

```

100 POKE 800,169: POKE 801,9
110 POKE 802,32: POKE 803,100: POKE 804,248
120 POKE 805,169: POKE 806,12
130 POKE 807,160: POKE 808,10
140 POKE 809,32: POKE 810,0: POKE 811,248
150 POKE 812,96

```

---

Enter and run that loader. After that, check it out with this BASIC program:

```

10 GR
20 CALL 800
30 END

```

Notice that the BASIC loader for this two-phase machine-language program is longer than the BASIC program that uses it. What's more, the entire machine-language program could be replaced with two simple BASIC statements, `COLOR=9` and `PLOT 10,12`. Though BASIC seems simpler now, you will later find you can do things with machine language that are almost impossible to do with BASIC.

**Drawing Horizontal Lines With HLINE** The HLINE routine at address -2023 draws a low-resolution horizontal line between two prescribed points on the screen, using a prescribed color code. It is the machine-language version of the BASIC HLINE statement.

Before running HLINE, the microprocessor must have the vertical position of the line in its A register, the left-hand horizontal coordinate in the Y register, and the right-hand horizontal coordinate in address location 44. So the general machine-language setup for the HLINE routine goes something like this:

```

LDA #REND      ;LOAD RIGHT COORDINATE TO REGISTER A
STA 44          ;STORE IT TO ADDRESS 42
LDA #VERT       ;LOAD VERTICAL POSITION TO REGISTER A
LDY #LEND       ;LOAD LEFT COORDINATE TO REGISTER Y
JSR -2023       ;EXECUTE THE MONITOR'S HLINE ROUTINE
RTS             ;AND RETURN TO BASIC

```

Just as with HLIN, the right-hand coordinate, left-hand coordinate, and vertical position can be integers in the range from 0 to 39.

Assuming that you are setting the color code from BASIC and would like to run such a routine from address 800, you would write the assembly version like so:

```

800 169 20      LDA #20      ;RIGHT COORDINATE OF 20
802 141 00 44   STA 42       ;STORE IT AT ADDRESS 42
805 169 10      LDA #10      ;VERTICAL POSITION OF 10
807 160 5       LDY #5       ;LEFT COORDINATE OF 5
809 32 25 248   JSR -2023    ;EXECUTE HLINE
812 96          RTS         ;RETURN TO BASIC

```

That particular routine is set up to draw a horizontal line of some prescribed color between horizontal coordinates 5 and 20 at vertical position 10. It amounts to doing a HLIN 5,20 AT 10 from BASIC.

Here is a suitable BASIC loader for that routine:

---

```

100 POKE 800,169: POKE 801,20
110 POKE 802,141: POKE 803,0: POKE 804,44
120 POKE 805,169: POKE 806,10
130 POKE 807,160: POKE 808,5
140 POKE 809,32: POKE 810,25: POKE 811,248
150 POKE 812,96

```

---

Enter and run that loader, then call it up at any later time with this sort of BASIC program:

---

```

10 GR
20 COLOR=4
30 CALL 800
40 END

```

---

This will draw the horizontal line using color code 4. If you want to change the right-hand coordinate of the line, POKE the desired value into address 801. If you want to change the left-hand coordinate, POKE the new value into address 808. Finally, if you want to change the vertical position, POKE the new value into address 806.

**Drawing Vertical Lines With VLINE** Just as the HLINE routine draws horizontal lines, the VLINE routine at address -2008 draws vertical lines. The necessary setup in this case is to get the top vertical coordinate into register A, the bottom vertical coordinate into RAM address 45, and the horizontal position into register Y, as shown here:

```
LDA #BEND ;BOTTOM COORDINATE TO REGISTER A
STA 45    ;STORE IT AT ADDRESS 45
LDA #TEND ;TOP COORDINATE TO REGISTER A
LDY #HORZ ;HORIZONTAL COORDINATE TO REGISTER Y
JSR -2008 ;EXECUTE MONITOR'S VLINE ROUTINE
RTS      ;RETURN TO BASIC
```

The BEND, TEND, and HORZ values can be anywhere from 0 to 30, just as long as BEND is greater than TEND. The BASIC equivalent of this is:

```
VLIN TEND,BEND AT HORZ
```

Use Tables 8-1, 8-2, and 8-3 to assemble that program (that is, convert it to machine language), beginning from address 800. Make up and check out a BASIC loader for it. The program will be quite similar to the one illustrated for HLINE in the previous section of this chapter.

**Getting Key Codes With RDKEY** A monitor routine located at address -756 prints the blinking cursor on the screen and waits for the user to make a keystroke. It loads the key code for the key into the A register, and goes on from there. Doing a CALL -756 directly from a BASIC program can be a useful trick whenever you want to halt operations until the user strikes any key; but the key code, itself, is lost when RDKEY is used that way.

Recall that the monitor routine COUT1 prints to the screen whatever character is represented in the A register. You can now write a short machine-language routine that calls RDKEY to fetch a key code to the A register, then calls COUT1 to print the character to the screen. The routine is:

```
800 32 12 253 JSR RDKEY ;GET A KEY CODE TO A
803 32 240 253 JSR COUT1 ;PRINT IT TO THE SCREEN
806 96      RTS      ;RETURN TO THE CALLING PROGRAM
```

The BASIC loader for this routine looks like this:

---

```
100 POKE 800,32: POKE 801,12: POKE 802,253
110 POKE 803,32: POKE 804,240: POKE 805,253
120 POKE 806,96
```

---

Enter and run that loader, then check its operation with this short BASIC program:

---

```
10 CALL -936
20 CALL 800
30 GOTO 20
```

---

You should find that you can type away on the screen to your heart's content and control the cursor's position. The program represents a full-screen text editor.

Table 8-5 summarizes the monitor routines that are featured in this chapter. The table begins by showing the name of the routine, its starting address, and the code sequence for executing it from your own machine-language routine. The accompanying comments describe what the routine does and the required setup procedures.

## PASSING VARIABLES TO A MACHINE-LANGUAGE ROUTINE

You might have noticed that all but the RDKEY routine described thus far in this chapter require some preliminary setup data. The COUT1 routine, for example, requires that the desired character code be loaded into register A. So far, you have been supplying this data by POKE-ing it to the X, Y, or A registers. But there is a better way to *pass a variable to* machine-language routines, and that is by passing it through a register of variables. This "register" occupies addresses 900 through 924, as shown in Fig. 8-1.

Suppose that you want to use the STOADV routine to print out the entire Apple character set. The idea is to pass the character codes one at a time and in succession to a custom STOADV setup routine. Here is what the machine-language portion of the program might look like:

```
LDA 900      ;LOAD REGISTER A WITH THE CONTENT OF 900
JSR STOADV   ;PRINT IT TO THE SCREEN
RTS          ;RETURN TO THE CALLING PROGRAM
```

Whatever character code happens to be in address 900 is thus printed to the screen by the subsequent STOADV routine. The idea is to POKE the de-

**Table 8-5. Monitor Routines Not Available to BASIC**

<b>Mnemonic</b>	<b>Machine Code</b>	<b>Definition</b>	<b>Requirement</b>
STOADV	-1040 32 240 251	Prints a character to the screen and advances the cursor	Load the character code to register A prior to calling this routine from machine language
COUT1	-528 32 240 253	Prints all but a control character to the screen and advances the cursor	Load the character code to register A prior to calling this routine from machine language
PRBL2	-1718 32 74 249	Prints between 1 and 256 consecutive blanks on the screen	Load the number of blanks to the X register prior to calling this routine from machine language
SETCOL	-1948 32 100 248	Sets the graphic color for low-resolution operations	Load the color code (0-15) to the A register prior to calling this routine from machine language
PLOT	-2048 32 0 248	Plots a low-resolution block of a prescribed color at graphic coordinates X and Y	Prior to calling this routine from machine language: Load the Y coordinate to register A Load the X coordinate to register Y

**Table 8-5—cont. Monitor Routines Not Available to BASIC**

<b>Mnemonic</b>	<b>Machine Code</b>	<b>Definition</b>	<b>Requirement</b>
HLINE	-2023    32 25 248	Plots a low-resolution horizontal line of a prescribed color	<p>Prior to calling this routine from machine language:</p> <p>    Load the ending X coordinate to address 44</p> <p>    Load the Y position to register A</p> <p>    Load the starting X coordinate to register Y</p>
VLINE	-2008    32 40 248	Plots a low-resolution vertical line of a prescribed color	<p>Prior to calling this routine from machine language:</p> <p>    Load the ending Y coordinate to address 45</p> <p>    Load the starting Y coordinate to register A</p> <p>    Load the X position to register Y</p>
RDKEY	-756     32 12 253	Prints the flashing cursor and waits for a key-stroke	<p>Executing this routine from machine language leaves the key code in register A</p>

sired character code to address 900 just prior to calling the machine-language routine.

You can assemble the machine-language routine this way:

```
800 173 132 3   LDA 900      ;FETCH CHARACTER TO A
803 32 240 251 JSR STOADV    ;PRINT IT
806 96                ;RETURN TO CALLING ROUTINE
```

and load it from BASIC this way:

---

```
100 POKE 800,173: POKE 801,132: POKE 802,3
110 POKE 803,32: POKE 804,240: POKE 805,251
120 POKE 806,96
```

---

The BASIC program that uses that routine can have this form:

---

```
10 CALL -936
20 FOR N=0 TO 255
30 POKE 900,N
40 CALL 800
50 NEXT N
60 END
```

---

Line 30 in that program passes the value of variable N to address 900 just before calling the subroutine in line 40. So as the BASIC portion of the program lets N cycle from 0 through 255, those values are passed to the machine-language portion of the program through address 900.

Table 8-6 shows the scratchpad portion of our custom memory map, addresses 900 through 924, broken down into *addrL* and *addrH* components. That should help you quite a bit when it comes to assembling machine-language routines that refer to those address locations.

The BASIC program in Listing 8-1 passes four different variables to a machine-language routine. The routine in this instance sets up and executes the low-resolution graphics SETCOL and HLINE. SETCOL requires a color code, so BASIC passes that code to the routine. HLINE requires an X starting point, an X ending point, and a Y coordinate; BASIC passes them to the machine-language routine as well.

Here is a line-by-line analysis of the BASIC portion of the program:

Line 10 goes to the loader subroutine at line 200.

Line 20 sets the TEXT mode and clears the screen.



**Table 8-6. High and Low Address Components of Scratchpad**

Address	<i>addrL</i>	<i>addrH</i>
900	132	3
901	133	3
902	134	3
903	135	3
904	136	3
905	137	3
906	138	3
907	139	3
908	140	3
909	141	3
910	142	3
911	143	3
912	144	3
913	145	3
914	146	3
915	147	3
916	148	3
917	149	3
918	150	3
919	151	3
920	152	3
921	153	3
922	154	3
923	155	3
924	156	3

Lines 30 and 40 INPUT the desired color code CC, and POKE it to scratchpad memory location 900.

Lines 50 and 60 INPUT the starting X coordinate as XSTRT and the Y coordinate as YSTRT, and POKEs them to memory locations 901 and 902, respectively.

Line 70 INPUTS the desired line length as LNTH.

Line 80 calculates the ending X coordinate by summing XSTRT and LNTH, and POKEs the result to address 903.

Lines 90 and 100 set the GR mode and call the machine-language sub-routine that draws the line on the screen.

## Listing 8-1. Passing Variables to a Machine-Language Routine.

---

```
10 GOSUB 200
20 TEXT : CALL -936
30 INPUT "WHAT COLOR CODE (0-15)",CC
40 POKE 900,CC
50 INPUT "WHAT STARTING COORDINATE (AS X,Y)",XSTRT,YSTRT
60 POKE 901,XSTRT: POKE 902,YSTRT
70 INPUT "HOW LONG",LNTH
80 POKE 903,XSTRT+LNTH
90 GR
100 CALL 800
110 PRINT "STRIKE ANY KEY TO DO AGAIN ..."
120 CALL -741: GOTO 20
200 POKE 800,173: POKE 801,132: POKE 802,3
210 POKE 803,32: POKE 804,100: POKE 805,248
220 POKE 806,173: POKE 807,135: POKE 808,3
230 POKE 809,141: POKE 810,44: POKE 811,0
240 POKE 812,173: POKE 813,134: POKE 814,3
250 POKE 815,172: POKE 816,133: POKE 817,3
260 POKE 818,32: POKE 819,25: POKE 820,248
270 POKE 821,96
280 RETURN
```

---

Lines 110 and 120 prompt the user to STRIKE ANY KEY TO DO AGAIN, use a CALL -741 to wait for any keystroke, then loop back to line 20 to give the user a chance to enter the parameters for drawing a different line.

Here is a memory map of the scratchpad memory for this program:

900— Variable CC, the color code.

901— Variable XSTRT, the starting X coordinate of the line.

902— Variable YSTRT, the Y position of the line.

903— Variable XEND, the sum of XSTRT and LNTH.

The machine-language portion of the program grabs those values as they are needed. Here is a combined machine-language and assembly-language version of that routine:

800	173	132	3	LDA CC	;FETCH CC FROM 900
803	32	100	248	JSR SETCOL	;SET THE COLOR
806	173	135	3	LDA XEND	;FETCH XEND FROM 903
809	141	44	0	STA 44	;AND STORE IT TO 44
812	173	134	3	LDA YSTRT	;FETCH YSTRT FROM 902
815	172	133	3	LDY XSTRT	;XSTRT FROM 901 TO REGISTER Y
818	32	25	248	JSR HLINE	;DRAW THE LINE WITH HLINE
821	96			RTS	;RETURN TO CALLING PROGRAM

---

## Listing 8-2. Passing Variables to Different Routines.

---

```
10 GOSUB 200
20 TEXT : CALL -936
30 INPUT "WHAT COLOR CODE (0-15)",CC
40 POKE 900,CC
50 INPUT "WHAT STARTING COORDINATE (AS X,Y)",XSTRT,YSTRT
60 POKE 901,XSTRT: POKE 902,YSTRT
70 INPUT "HOW LONG",LNTH
80 INPUT "HORIZONTAL OR VERTICAL (H/V)?",DIR$
90 IF DIR$="V" THEN 140
100 IF DIR$#"H" THEN 80
110 POKE 903,XSTRT+LNTH
120 GR : CALL 800
130 GOTO 160
140 POKE 903,YSTRT+LNTH
150 GR : CALL 822
160 PRINT "STRIKE ANY KEY TO DO AGAIN ..."
170 CALL -741: GOTO 20
200 POKE 800,173: POKE 801,132: POKE 802,3
210 POKE 803,32: POKE 804,100: POKE 805,248
220 POKE 806,173: POKE 807,135: POKE 808,3
230 POKE 809,141: POKE 810,44: POKE 811,0
240 POKE 812,173: POKE 813,134: POKE 814,3
250 POKE 815,172: POKE 816,133: POKE 817,3
260 POKE 818,32: POKE 819,25: POKE 820,248
270 POKE 821,96
280 POKE 822,173: POKE 823,132: POKE 824,3
290 POKE 825,32: POKE 826,100: POKE 827,248
300 POKE 828,173: POKE 829,135: POKE 830,3
310 POKE 831,141: POKE 832,45: POKE 833,0
320 POKE 834,173: POKE 835,134: POKE 836,3
330 POKE 837,172: POKE 838,133: POKE 839,3
340 POKE 840,32: POKE 841,40: POKE 842,248
350 POKE 843,96
360 RETURN
```

---

Notice how it fetches the necessary variables from the little scratchpad memory as they are needed for setting up and executing the SETCOL and HLINE routines.

The BASIC listing in Listing 8-2 works much the same way, but gives the user the option of selecting a horizontal or vertical line. This listing illustrates the technique for passing variables to a machine-language routine, and the notion of calling one of two different machine-language routines from a BASIC main program.

Here is a detailed analysis of Listing 8-2:

Line 10 goes to the machine-language loader routine at line 200.

Line 20 sets the TEXT mode and clears the screen.

Lines 30 and 40 INPUT the desired color code as variable CC, and POKE it to scratchpad memory address 900.

Lines 50 and 60 INPUT the starting X and Y coordinates as variables XSTRT and YSTRT, and POKE them to addresses 901 and 902, respectively.

Line 70 INPUTs the desired line length as variable LNTH.

Line 80 INPUTs the desired line direction (H or V) as string variable DIR\$.

Line 90 goes to line 140 if the user specifies a vertical line.

Line 100 goes back to line 80 to INPUT DIR\$ again if the direction is neither H nor V.

Lines 110 through 130 set up and execute the routine for drawing a horizontal line. They POKE the ending X coordinate (XEND) to address 903 as the sum of XSTRT and LNTH, set the GR mode, CALL the horizontal-drawing machine-language routine at address 800, and go to line 160 to prompt the user's next move.

Lines 140 and 150 set up and execute the routine for drawing a vertical line. They POKE the ending Y coordinate (YEND) to address 903 as the sum of YSTRT and LNTH, set the GR mode, and CALL the vertical-drawing portion of the machine-language routine at address 822.

Lines 160 and 170 prompt the user to STRIKE ANY KEY TO DO AGAIN, wait for a keystroke, and return to line 20 to start the drawing INPUT routines again.

Here is the scratchpad memory map:

900—Variable CC, the color code.

901—Variable XSTRT, the starting X coordinate for a horizontal line, or the X position for a vertical line.

902—Variable YSTRT, the starting Y coordinate for a vertical line, or the Y position for a horizontal line.

903—XEND (the sum of XSTRT and LNTH) for a horizontal line, YEND (the sum of YSTRT and LNTH) for a vertical line.

The machine-language portion of the routine is divided into two separate parts: one for setting up and executing a horizontal-line drawing routine, and another for setting up and executing a vertical-line drawing routine. The part that draws horizontal lines begins at RAM address 800, and the portion that draws vertical lines begins at 822. So, to set up and draw a horizontal line, the BASIC program should CALL 800. To set up and draw a vertical line, the BASIC program should CALL 822. The addresses 800 and 822 are the *entry points* for this machine-language routine:

```

800 173 132 3   LDA CC      ;FETCH CC FROM 900
803 32 100 248 JSR SETCOL   ;SET THE COLOR
806 173 135 3   LDA XEND    ;FETCH XEND FROM 903
809 141 44 0    STA 44      ;STORE IT TO ADDRESS 44
812 173 134 3   LDA YSTRT   ;FETCH YSTRT FROM 902
815 172 133 3   LDY XSTRT   ;XSTRT FROM 901 TO REGISTER Y
818 32 25 248   JSR HLINE    ;DRAW LINE FROM HLINE ROUTINE
821 96          RTS         ;RETURN TO CALLING ROUTINE
822 173 132 2   LDA CC      ;FETCH CC FROM 900
825 32 100 248 JSR SETCOL   ;SET THE COLOR
828 173 135 3   LDA YEND    ;FETCH YEND FROM 903
831 141 45 0    STA 45      ;AND STORE IT TO ADDRESS 45
834 173 134 3   LDA YSTRT   ;FETCH YSTRT FROM 902
837 172 133 3   LDY XSTRT   ;XSTRT FROM 901 TO REGISTER Y
840 32 40 248   JSR VLINE    ;DRAW LINE FROM VLINE ROUTINE
843 96          RTS         ;RETURN TO CALLING ROUTINE

```

## PASSING VARIABLES FROM A MACHINE-LANGUAGE ROUTINE

RDKEY is a commonly used monitor routine that waits for a keystroke from the keyboard. When that single keystroke occurs, the key code is placed into register A. BASIC cannot get directly to any of the registers in the microprocessor, but it can get to that information in an indirect fashion—by a routine that passes the key code through a designated RAM location.

You should be aware of the fact that executing a CALL -741 from BASIC causes the system to wait for *any* keystroke to occur. Suppose, however, that you want the system to wait for a *particular* keystroke to occur. You can CALL RDKEY from BASIC, wait for the keystroke, pass the content of the A register back to BASIC, test the value and take appropriate action from there.

Consider this simple routine:

```

800 32 12 253   JSR RDKEY    ;WAIT FOR A KEYSTROKE
803 141 132 3   STA 900      ;SAVE KEY CODE IN 900
806 96          RTS         ;RETURN TO CALLING ROUTINE

```

It simply executes the monitor's RDKEY routine. When the keystroke occurs, the key code in register A is passed to scratchpad address 900. Listing 8-3 loads the routine and uses it from BASIC.

Enter this program, run it, and follow the prompting messages. As far as the BASIC portion of the routine is concerned, a CALL 800 causes the system to wait for a keystroke. When that keystroke occurs, the corresponding key code is found in address 900.

### Listing 8-3. Passing Variables From a Machine-Language Routine.

---

```
10 GOSUB 200
20 CALL -936
30 PRINT "STRIKE SPACE BAR TO CONTINUE..."
40 CALL 800
50 IF PEEK (900)<>160 THEN 40
60 PRINT "IT WORKED!!"
70 PRINT
80 PRINT "WANT TO DO AGAIN (Y/N)?"
90 CALL 800
100 IF PEEK (900)=217 THEN 20
110 IF PEEK (900)=206 THEN END
120 GOTO 90
200 POKE 800,32: POKE 801,12: POKE 802,253
210 POKE 803,141: POKE 804,132: POKE 805,3
220 POKE 806,96
230 RETURN
```

---

Line 40, for instance, calls this subroutine and returns with the key code at address 900. Line 50 in the program tests that value by PEEKing into 900 and comparing it with 160—the key code for a space-bar keystroke. If the value in 900 is *not* 160, the BASIC program loops back to line 40 to check the keyboard again. Line 60, in other words, is not executed until the user strikes the space bar.

The BASIC program calls the machine-language routine again in line 90. This time, however, the program is looking for either a Y (key code 20) or an N (key code 206). Line 120 handles any other keystroke by looping back to line 90 to call the machine-language routine again.

It is possible to do the same sort of thing without resorting to a custom machine-language routine; a PEEK(-16384) strobes the keyboard in a similar fashion and can assign the key code to any BASIC variable. That approach, however, requires resetting the keyboard strobe by doing a POKE -16368,0. What's more, PEEK doesn't print a flashing cursor on the screen.

So this is an instance where a short machine-language routine can make a common programming situation run a bit simpler and smoother. Bear in mind, though, that we are using it here as an example of passing a variable from machine language to BASIC.

# The Memory Environment

All personal computers have two kinds of memory: random-access memory (RAM) and read-only memory (ROM). The two differ in that RAM may be read from or written to, but ROM may only be read. Turning off the computer destroys data saved in RAM. Data in ROM, on the other hand, is stored permanently. Therefore, turning off the system doesn't affect the ROM. POKEing to ROM has no effect, either.

Incidentally, 255 is the largest decimal number that can be POKEd into a RAM address location. Try POKEing a number larger than 255 and some entirely different value will be stored there. Having a 1-byte limit on the size of numbers that can be stored in RAM means that POKEing anything besides integer values from 0 to 255 will cause problems.

The memory system is organized by addresses. Most of those addresses are devoted to RAM and ROM devices, but there is a handful that are used for *memory-mapped* I/O functions. We will describe these RAM, ROM, and I/O addresses in this chapter, because knowing how memory is organized and how it is used can help you set up better programs and avoid some frustrating bugs.

The discussions also introduce hexadecimal notation. (If you aren't sure about how to convert decimal to hexadecimal or hexadecimal back into decimal notation, take some time to study Appendix A.) In keeping with the Apple/6502 notation conventions, all hexadecimal notations are preceded by a dollar sign; unless that dollar sign is shown, you can rightly conclude that the value is in decimal form.

**LOWER RAM ADDRESSES \$0000 THROUGH \$0BFF** These addresses are common to all systems, regardless of the amount of extra RAM they might have. The very lowest address is \$0000 (decimal 0) and the highest address in this particular case is \$0BFF (or 3071 decimal). Fig. 9-1 is an overall memory map of this RAM area.

Notice that the figure shows the *starting* address of each area. For instance, the keyboard input buffer area starts at \$0200, and ends at \$02FF,

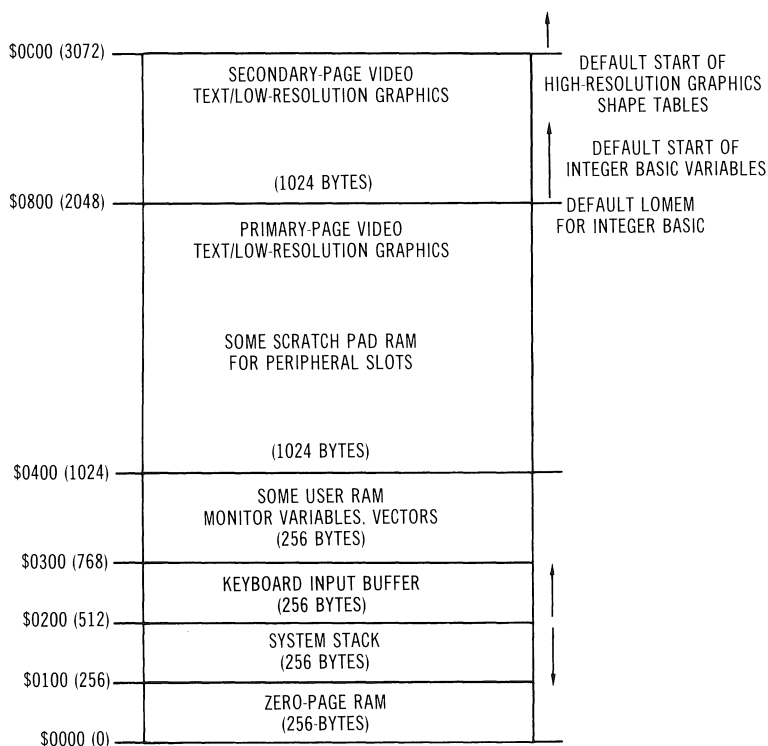


Fig. 9-1. Lower memory map.

or decimal 1021. Knowing where these areas are located will help you plan your use of RAM. In this way, your programs will not compete for the same RAM space that the system uses.

**Zero-Page RAM: \$0000–\$00FF** The location of the zero-page RAM area is fixed by the engineering of the 6502 microprocessor, and there is no way to change it. The Apple monitor and Integer BASIC use this area quite extensively for saving and retrieving important bytes of data, but you can use it to some advantage. You have to know how the monitor and Integer BASIC use the area and where the unused addresses are located.

This zero-page area is so important, that the 6502 instruction set includes a number of instructions that refer exclusively to it. In these instructions, the zero-page addressing is shortened to just two hexadecimal characters—the notation omits the two leading zeros. You can still refer to addresses in the zero-page area with the standard four-character format, but two characters will suffice.



Thus, zero-page address \$1A is the same as address \$001A. The former uses fewer characters; if it is written into a machine-language program, it takes up just one byte of memory as opposed to two bytes for the conventional four-character specification. Remember, though, that abbreviated addressing applies only to the zero-page RAM area. All other addresses in the system must be specified with the four-character format. (Addresses in decimal form can always be specified with the leading zeros omitted.)

Fig. 9-2 shows the details of the zero-page RAM area. All 256-byte locations are shown, from 0 to 255. The actual address of a particular byte is found by summing the decimal or hexadecimal value along the left side of the diagram with the corresponding decimal or hexadecimal value along the top. Address \$55, or decimal 85, is the last monitor byte that is used in the zero-page area, for example. The first location used exclusively for Integer BASIC is at address \$56, or 86.

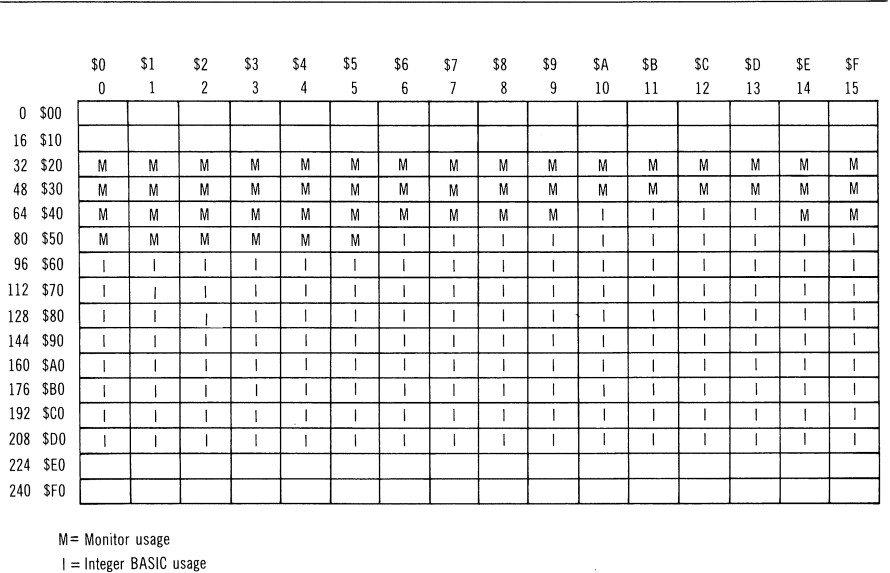


Fig. 9-2. Zero-page memory map.

The diagram clearly shows two areas of zero-page memory that aren't used by either the monitor or Integer BASIC. The first section extends from address \$00 through \$1F, and the second is from \$E0 to \$FF. Some of those "unused" addresses are actually used when you first turn on the computer or boot up DOS. But generally they are free for custom applications after the system is up and running.

If you are working exclusively with machine-language programs, you are also free to use the addresses designated for Integer BASIC. There is no way, however, to get away from the system monitor—it is always at work in the Apple.

While it is important to avoid using zero-page addresses that are normally assigned to the monitor and Integer BASIC, you can work with the addresses as the system defines them. You were doing that when you were setting up a custom text window by POKEing values for WNDLFT, WNDWIDTH, WNDTOP, and WNDBTM into the appropriate addresses in zero-page RAM. Table 9-1 shows which addresses are available for your applications, and which addresses are used by the monitor and Integer BASIC and what they mean. The table doesn't account for all zero-page addresses used for the monitor and Integer BASIC, but rather spells out the purpose of those deemed most useful.

**Table 9-1. Zero-Page Addresses**

Address	Mnemonic	Definition
\$20 (32)	WNDLFT	Column address of the left-hand edge of the primary-page text window. Range is \$00–\$27 (0–39). Normal value is \$00 (0).
\$21 (33)	WNDWIDTH	Number of characters in each line of video text. Range is \$01–\$28 (1–40). Normal value is \$28 (40).
\$22 (34)	WNDTOP	Row address of the top line of video text. Range is \$00–\$16 (0–22). Normal value for full-screen text is \$00 (0); for mixed text/graphics it's \$14 (20).
\$23 (35)	WNDBTM	Number of text window lines plus the content of WNDTOP. Range is \$01–\$18 (1–24). Normal value is \$18 (24).
\$24 (36)	CH	Current horizontal displacement of the text cursor

**Table 9-1—cont. Zero-Page Addresses**

Address	Mnemonic	Definition
\$25 (37)	CV	from WNDLFT. Range is 0 to WNDWDTH) minus 1.  Current vertical displacement of the text cursor relative to WNDTOP. Range is \$00–\$17 (0–23).
\$26,\$27 (38,39)	GBASL and GBASH	Low- and high-order bytes of the video address of the current cursor line for low-resolution graphics. Serves the same purpose for high-resolution graphics as HBASL and HBASH.
\$28,\$29 (40,41)	BASL and BASH	Low- and high-order bytes of the video text address for the current cursor line.
2C (44)	H2	Right endpoint of a low-resolution horizontal line being drawn by the HLINE function.
\$2D (45)	V2	Bottom endpoint of a low-resolution vertical line being drawn by the VLINE function.
\$2E (46)	MASK	Selects whether a low-resolution color block is plotted in the upper or lower half of a screen location. Carries a value of \$0F (15) for plotting in the lower half; \$F0 (240) for plotting in the upper half.
\$30 (48)	COLOR	Low-resolution graphics color code.

**Table 9-1—cont. Zero-Page Addresses**

Address	Mnemonic	Definition
\$32 (50)	INVFLG	Video text format register; use \$FF (255) for normal white-on-black, \$7F (127) for flashing text, or \$3F (63) for inverse black-on-white.
\$33 (51)	PROMPT	Text character code for INPUT prompt.
\$4A,\$4B (74,75)	LOMEML and LOMEMH	Low- and high-order byte for the current LOMEM address.
\$4C,\$4D (76,77)	HIMEML and HIMEMH	Low- and high-order byte for the current HIMEM address.
\$4E,\$4F (78,79)	RNDL and RNDH	Two-byte number that increments rapidly during several KEYIN-type monitor routines. The purpose is to provide a random number for many kinds of programming applications.
\$CA,\$CB (202,203)	BASSTL and BASSTH	Low- and high-order bytes for the current address of Integer BASIC programming. Usually set to HIMEM if there is no BASIC programming.
\$CC,\$CD (204,205)	BASVRL and BASVRH	Low- and high-order bytes for the current address of variables stored by Integer BASIC. Usually set to LOMEM if there are no variables in the list.

**Table 9-1—cont. Zero-Page Addresses**

Address	Mnemonic	Definition
\$E0,\$E1 (224,225)	HIRESXL and HIRESXH	Low- and high-order bytes of the current high-resolution X coordinate.
\$E2 (226)	HIRESV	Current high-resolution Y coordinate.
\$E6 (230)	HPAG	High-resolution page for plotting; use \$20 (32) for primary page; \$40 (64) for secondary page.
\$E8,\$E9 (232,233)	SHPTR	Low- and high-order bytes of the beginning of the high-resolution shape tables.

**System Stack RAM: \$0100–\$01FF** Like the zero-page RAM area, the location of the system stack is fixed—there is no way to alter it. However, the system does not fully use the stack in normal operations. This means that you can use some of the stack for your own machine-language programs.

Since the system uses the stack at address \$01FF and works downward from there, the space that you may use begins at the very bottom of the stack, at \$0100. Normally, you can save data at addresses \$0100 to \$010F without fear of interfering with stack operations. However, you shouldn't use more stack than that. To avoid all possibility of contention with the system for RAM space, don't use the stack at all.

**The Keyboard Input Buffer: \$0200–\$02FF** This 256-byte area of RAM is used for storing character codes that are entered from the keyboard under normal monitor and Integer BASIC conditions. The area is not fixed by the 6502 microprocessor, but rather by the original Apple engineers. With some difficulty, you can write a machine-language keyboard routine that puts the keyboard input buffer somewhere else in RAM. It usually isn't worth all the trouble, though.

Any time you type in a command or respond to an INPUT statement from the keyboard, those key codes are stacked in a bottom-up fashion in the keyboard input buffer. They continue building up, one address at a

time, until you strike the RETURN key. If you are in a command mode of operation, the system then reads the contents of the keyboard input buffer and takes the appropriate action. If you are responding to an INPUT statement in a BASIC program, striking the RETURN key causes the contents of the keyboard input buffer to be assigned to the designated numeric or string variable.

The following Integer BASIC program allows you to type some characters into the keyboard buffer and then dump them to the video text screen. Load the program and respond to the prompt symbol and flashing cursor by entering some arbitrary string of keyboard characters. The instant you end that entry process by striking the RETURN key, the program displays the content of the entire keyboard buffer. Notice that your string of characters is reproduced at the top of the screen and that it always ends with an M character. It is that M character (actually a representation of the RETURN key entry) that the system uses to mark the end of the current keyboard buffer entry.

---

```
10 CALL -936
20 CALL -662
30 CALL -936
40 FOR N=0 TO 255
50 POKE 256* PEEK (41)+ PEEK (40)+ PEEK (36), PEEK (512+N)
60 CALL -1036
70 NEXT N
80 VTAB 10: TAB 1
90 GOTO 20
100 XX=YY=COLR
110 INIT=-12288:BKGND=-11471
120 POSN=-11527:PLOT=-11506:LINE=-11500
130 CLEAR=-12274
```

---

Here is how it works:

Line 10 homes the cursor and clears the screen.

Line 20 calls the GETLIN monitor routine to enter keystrokes into the keyboard buffer. GETLIN returns control to BASIC when the RETURN key is struck.

Line 30 homes the cursor and clears the screen.

Lines 40 through 70 display the contents of the keyboard buffer, from addresses 512 through 767.

Lines 80 and 90 get the cursor symbol out of the way, and return to line 20 to input another series of keystrokes.

---

The series of characters that you type into the buffer ends with that M character. The rest of the data in the keyboard buffer is garbage. It consists of previously typed material. To illustrate this, enter a rather long series of characters followed by a relatively short series. You will see the short message at the beginning of the buffer and the remainder of the long message appearing after the message-ending M character.

Knowing how to get at the data in the keyboard buffer can be useful when writing BASIC or machine-language programs that call for dumping the result of a keyboard operation into some prescribed portion of user RAM.

**Variables, Vectors, and User RAM: \$0300–\$03FF** Like the zero-page RAM area, this is one of those segments of RAM that can be used for custom programming of a limited sort. The only problem with using this area for your own purposes is that it sometimes conflicts with special monitor and BASIC operations.

For the most part, the lower addresses are used by the system only during initial start-up operations: booting up DOS, for example. Addresses \$0320 through \$032A are frequently used for high-resolution graphics routines, especially those referring to the Programmer's Aid package. And DOS makes extensive use of addresses \$0399–\$03EA. Addresses through the remainder of this RAM area are used for monitor *vectors*, or addresses that point to other machine-language subroutines.

The safest working area for custom machine-language programming is thus at the lower end of the area, from \$0300 through \$0320. But if you aren't using the special high-resolution graphics functions, you can use addresses up to \$0399.

If you find that your programs written into this area are “blowing up” unaccountably, you have probably stumbled across a conflict of usage. The best thing to do in that case is to look for another segment of RAM for your program.

**Primary-Page Text/Graphics: \$0400–\$07FF** Earlier discussions of text and low-resolution graphics began with a description of this video RAM area. Table 9-2 outlines the video portion of that area again, but with the addition of hexadecimal addresses. Table 9-3 summarizes the small sections of RAM in the primary-page area that are used as a peripheral slot scratchpad rather than as video RAM.

In theory, it is altogether possible to enter custom machine-language programs into this area of RAM. But it isn't a good idea in practice, because the programming will be rendered as confusing garbage on the screen. It is better to use the video RAM area for its intended purpose. You can, however, use the small segments of scratchpad RAM as long as there

are no peripheral cards plugged into the locations. Again, the idea is to avoid possible conflict of RAM usage.

**Table 9-2. Primary-Page Video Addresses**

Video Line	Hex Address	Decimal Address
0	\$0400-\$0427	1024-1063
1	\$0480-\$04A7	1152-1191
2	\$0500-\$0527	1280-1319
3	\$0580-\$05A7	1408-1447
4	\$0600-\$0627	1536-1575
5	\$0680-\$06A7	1664-1703
6	\$0700-\$0727	1792-1831
7	\$0780-\$07A7	1920-1959
8	\$0428-\$044F	1064-1103
9	\$04A8-\$04CF	1192-1231
10	\$0528-\$054F	1320-1359
11	\$05A8-\$05CF	1448-1487
12	\$0628-\$064F	1576-1615
13	\$06A8-\$06CF	1704-1743
14	\$0728-\$074F	1832-1871
15	\$07A8-\$07CF	1960-1999
16	\$0450-\$0477	1104-1143
17	\$04D0-\$04F7	1232-1271
18	\$0550-\$0577	1360-1399
19	\$05D0-\$05F7	1488-1527
20	\$0650-\$0677	1616-1655
21	\$06D0-\$06F7	1744-1783
22	\$0750-\$0777	1872-1911
23	\$07D0-\$07F7	2000-2039

**Secondary-Page Text/Graphics: \$0800-\$0BFF** As mentioned in a preceding paragraph, it is possible, but rarely advisable, to write machine-language programs into primary-page video RAM. It is likewise possible, and often desirable, to write programs and data into secondary-page video RAM. In fact, even Integer BASIC uses secondary-page video RAM for storing data. You will find that a good many commercially available programs for the Apple load into this area. That's fine as long as you have no intention of using it for secondary-page video applications. If you avoid secondary-page video operations, you will have a full 1024 bytes of RAM for programming purposes. Apple users who have the smaller 4K



**Table 9-3. Peripheral Slot Scratchpad RAM**

Byte Number	SLOT NUMBER							
	0	1	2	3	4	5	6	7
0	\$0478 1144	\$0479 1145	\$047A 1146	\$047B 1147	\$047C 1148	\$047D 1149	\$047E 1150	\$047F 1151
1	\$04F8 1272	\$04F9 1273	\$04FA 1274	\$04FB 1275	\$04FC 1276	\$04FD 1277	\$04FE 1278	\$04FF 1279
2	\$0578 1400	\$0579 1401	\$057A 1402	\$057B 1403	\$057C 1404	\$057D 1405	\$057E 1406	\$057F 1407
3	\$05F8 1528	\$05F9 1529	\$05FA 1530	\$05FB 1531	\$05FC 1532	\$05FD 1533	\$05FE 1534	\$05FF 1535
4	\$0678 1656	\$0679 1657	\$067A 1658	\$067B 1659	\$067C 1680	\$067D 1681	\$067E 1682	\$067F 1683
5	\$06F8 1784	\$06F9 1785	\$06FA 1786	\$06FB 1787	\$06FC 1788	\$06FD 1789	\$06FE 1790	\$06FF 1791
6	\$0778 1912	\$0779 1913	\$077A 1914	\$077B 1915	\$077C 1916	\$077D 1917	\$077E 1918	\$077F 1919
7	\$07F8 2040	\$07F9 2041	\$07FA 2042	\$07FB 2043	\$07FC 2044	\$07FD 2045	\$07FE 2046	\$07FF 2047

RAM systems are often forced to do that to get a sufficient amount of working RAM.

Tables 9-4 and 9-5 show the complete memory map for this part of RAM. Most of it is devoted to secondary-page video, but there are small segments that aren't. Unlike the primary-page video section, these short segments of RAM serve no particular purpose. They are open for your own use.

**UPPER RAM ADDRESSES \$0C00 THROUGH \$BFFF**      The extent of the upper RAM area depends on how much RAM you have installed in your Apple. Even the very smallest memory scheme—one having just 4K (4096) bytes of RAM—includes the lower RAM area just described plus an upper 1K (1024) bytes for user-generated programs. Unfortunately for such users, there is no RAM space available for high-resolution graphics.

**Table 9-4. Secondary-Page Video Addresses**

<b>Video Line</b>	<b>Hex Address</b>	<b>Decimal Address</b>
0	\$0800–\$0827	2048–2087
1	\$0880–\$08A7	2176–2215
2	\$0900–\$0927	2304–2343
3	\$0980–\$09A7	2432–2471
4	\$0A00–\$0A27	2560–2599
5	\$0A80–\$0AA7	2688–2727
6	\$0B00–\$0B27	2816–2855
7	\$0B80–\$0BA7	2944–2983
8	\$0828–\$084F	2088–2127
9	\$08A8–\$08CF	2216–2255
10	\$0928–\$094F	2344–2383
11	\$09A8–\$09CF	2472–2511
12	\$0A28–\$0A4F	2600–2639
13	\$0AA8–\$0ACF	2728–2767
14	\$0B28–\$0B4F	2856–2895
15	\$0BA8–\$0BCF	2984–3023
16	\$0850–\$0877	2128–2167
17	\$08D0–\$08F7	2256–2295
18	\$0950–\$0977	2384–2423
19	\$09D0–\$09F7	2512–2551
20	\$0A50–\$0A77	2640–2679
21	\$0AD0–\$0AF7	2768–2807
22	\$0B50–\$0B77	2896–2935
23	\$0BD0–\$0BF7	3024–3063

**Table 9-5. Unused RAM in Secondary-Page Area**

<b>Segment</b>	<b>Hex Address</b>	<b>Decimal Address</b>
0	\$0878–\$087F	2168–2177
1	\$08F8–\$08FF	2296–2303
2	\$0978–\$097F	2424–2431
3	\$09F8–\$09FF	2552–2559
4	\$0A78–\$0A7F	2680–2687
5	\$0AF8–\$0AFF	2808–2815
6	\$0B78–\$0B7F	2936–2943
7	\$0BF8–\$0BFF	3064–3051

The present discussion deals with the upper RAM area for the larger Apple systems, those having 16K, 32K, or 48K of RAM. Such systems have enough memory for at least one page of high-resolution graphics and some extra space for other programming applications.

The most important feature of the memory maps for these regions is the way Integer BASIC uses them.

Integer BASIC programs begin at the HIMEM address and build *downward* from there.

Variable tables begin at the LOMEM address and build *upward* from there.

Whenever you are using Integer BASIC (or Applesoft BASIC, although it loads differently) you can count on the programming using RAM addresses between the LOMEM and HIMEM settings. The idea is to avoid putting custom machine-language programs anywhere in that area if you are using them in conjunction with BASIC programs. If you are not using any BASIC programming, the LOMEM and HIMEM settings have little relevance.

In instances where you do initialize Integer BASIC, the Apple system automatically sets the LOMEM address to \$0800, or 2048, and it sets HIMEM at the upper end of the available RAM space: \$4000 (16384) for a 16K system, \$8000 (32768) for a 32K system, and \$C000 (49152) for a 48K system.

You can, of course, change the LOMEM and HIMEM settings once the system is up and running. The significance of that procedure is that it lets you limit the RAM space that is used by BASIC programming, allowing you to use what's left for custom machine-language programming and similar applications.

**16K Systems** Fig. 9-3 outlines the upper RAM section for a 16K Apple system. It includes the secondary page of text/low-resolution graphics, already described in some detail on page 234, and extends through the primary page of high-resolution graphics to RAM address \$3FFF, or 16383.

There are just three main sections:

1. Secondary-page text/low-resolution graphics at RAM addresses \$0800 through \$0BFF (2148 through 3071).
2. User's RAM at addresses \$0C00 through \$1FFF (3072 through 8191).
3. Primary-page, high-resolution graphics video at addresses \$2000 through \$3FFF (8192 through 16383).

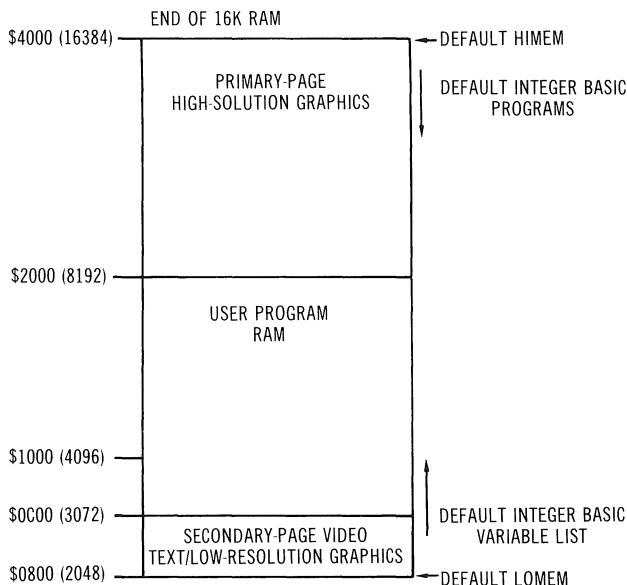


Fig. 9-3. Upper memory map for 16K systems.

Notice especially the default settings for LOMEM, HIMEM, and the high-resolution shape tables. Those default addresses are the ones the system sets up for you unless you specify otherwise.

Of special concern to users of 16K systems is the fact that HIMEM is set at the top of the high-resolution graphics video area. Unless you set HIMEM to the lower end of that area—to \$2000, or 8192—any BASIC programming is going to be inserted into the high-resolution graphics area. *Attempting to use high-resolution graphics from Integer BASIC without first setting HIMEM to 8192 will mess up both the program and the graphics.*

Furthermore, the system automatically sets the LOMEM address at the beginning of the secondary page of text/low-resolution graphics. If you attempt to use the secondary page of text/low-resolution graphics without first setting LOMEM to \$0C00 (3072) or higher, variable data from Integer BASIC will mess up the graphics.

The worst-case situation is where you want to use Integer BASIC, the secondary page of text/low-resolution graphics, and high-resolution graphics at the same time. The LOMEM and HIMEM addresses just cited solve the problem.

There are a number of different ways to set up the system in that fashion. From the Integer BASIC command mode:

HIMEM:8192  
LOMEM:3072

From the monitor command mode:

4A:00 0C 00 20

At the beginning of an Integer BASIC program:

POKE 74,0:POKE 75,12:POKE 76,0:POKE 77,32

As a machine-language routine:

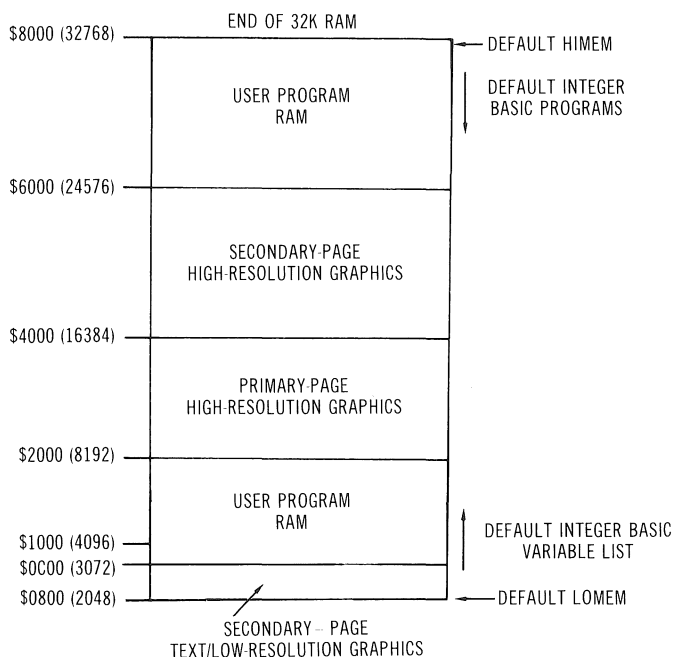
LDA #0  
STA 74  
STA 76  
LDA #12  
STA 75  
LDA #32  
STA 77

There is one remaining difficulty that cannot be resolved on a 16K system—the lack of a secondary-page, high-resolution graphics buffer, or memory. There simply isn't enough RAM space available.

**32K Systems** A 32K system has RAM available from address \$0000 through \$7FFF (0–32767). The lower portion of that area is used as described earlier. What is of special importance to the current discussion is the layout of the memory map for addresses \$0800 through \$7FFF (2048 through 32767). (See Fig. 9-4.)

The memory map shows five main sections:

1. Secondary-page text/low-resolution graphics at RAM addresses \$0800 through \$0BFF (2148 through 3071).
2. User's RAM at addresses \$0C00 through \$1FFF (3072 through 8191).
3. Primary-page, high-resolution graphics video at addresses \$2000 through \$3FFF (8192 through 16383).
4. Secondary-page, high-resolution graphics at addresses \$4000 through \$5FFF (16384 through 24575).
5. User's RAM at addresses \$6000 through \$7FFF (24576 through 32767).



**Fig. 9-4.** Upper memory map for 32K systems.

The default address for HIMEM is \$8000, or 32768. That means that Integer BASIC programs will begin at that address and build downward from there. Fortunately, there are no competing uses for that upper end of the RAM area. In fact, there is about 8K of programming RAM available before BASIC begins encroaching on the upper end of the secondary-page area for high-resolution graphics. (If a BASIC program has to be more than 8K-bytes long, you simply have to give up the secondary page of high-resolution graphics).

The default LOMEM address, as usual, is set at \$0800 (2048). That overlays the secondary-page text/low-resolution graphics area, so if you are planning to use secondary-page text or graphics with Integer BASIC, you ought to set LOMEM to a higher address, such as \$1000 (4096).

The worst-case 32K programming situation is where you want to use the full complement of graphics (both pages of low- and high-resolution graphics) along with a combination of Integer BASIC and custom machine-language programs. Resetting LOMEM can put things in order for you. The only question is where to set it.

If you choose to set LOMEM to \$1000, or 4096, you will have plenty of RAM for BASIC programming, but little room for high-resolution shape

tables and machine-language programming. If the machine-language programs are short ones, you can try using the RAM area from \$0300 to \$03AF as described in an earlier section, and leave \$0C00 through \$0FFF for shape tables.

If your programming situation calls for full-capability graphics, long machine-language programs or data files, and relatively short Integer BASIC programs, you can set LOMEM at the top of the secondary page of high-resolution graphics—to \$6000, or 24576. That confines BASIC programming to the upper portion of user RAM, and allows you to use the entire lower portion of user RAM for shape tables and machine-language programs. You can, for example, load machine-language programs from \$1000 to \$1FFF (4096 to 8191).

You might become a bit pinched for RAM in the upper user space if you are using DOS and Integer BASIC. They share that upper region, and you should consult your DOS manual to see where it loads.

**48K Systems** Users of 48K Apple systems have a very generous amount of RAM available for programming applications. The usable RAM area begins at \$0000 and extends through \$BFFF (0 through 49151). (See the upper portion mapped for you in Fig. 9-5. The lower section was described in detail earlier in this chapter.)

The only thing that might be a nuisance in some instances is the default setting of LOMEM. A lot of programmers like to begin loading machine-language programs from address \$0800 (2040). Protecting that area from BASIC is a simple matter of resetting LOMEM to some higher address such as \$6000, or 24576.

Setting LOMEM to \$6000 leaves all four graphics pages available and allows some 24K in the upper RAM area for BASIC programming. (DOS uses some of that upper RAM area, too, but there is so much space available that there is generally no need to give it any special thought.)

**An Overall Summary** If you do not intend to use any BASIC programming at all, you have the widest choice of RAM area available for custom machine-language programming. You need not worry about LOMEM and HIMEM settings at all. In fact, they aren't even set until you do that CTRL-B command that initializes the Integer BASIC system.

If you have a 16K system, you often have to make some trade-offs regarding the size of the program and the use of secondary-page text/low-resolution graphics or primary-page high-resolution graphics. The 32K and 48K systems, on the other hand, have enough RAM available for long machine-language programs and full graphics capability.

The time to take special care is when you are setting up combinations of full-capability graphics, machine language, and BASIC. Then it is gen-

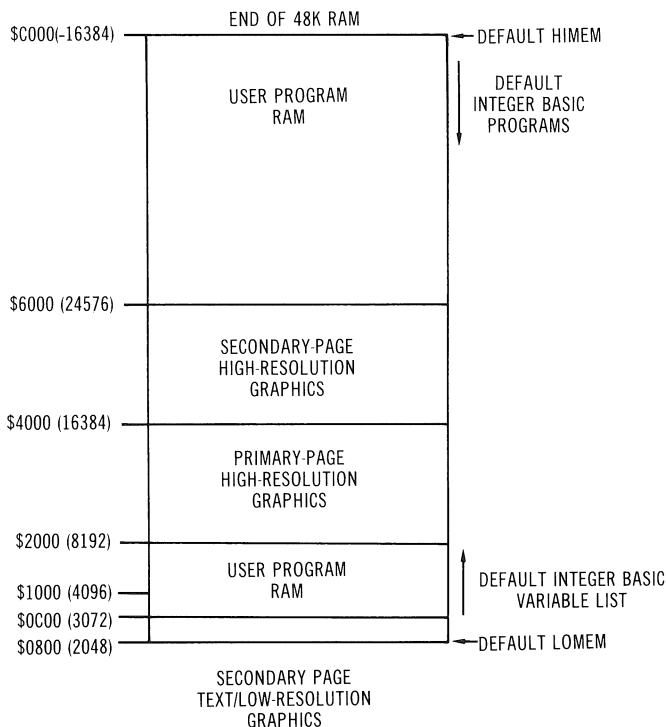


Fig. 9-5. Upper memory map for 48K systems.

erally just a matter of setting LOMEM to a higher address, thereby pinching BASIC into a smaller RAM area and protecting your machine-language programming in the lower RAM segments.

It's all a matter of knowing how the Apple's RAM is organized, where the default addresses are for your particular system, and how to tinker with LOMEM and HIMEM to get the custom memory map you want. Taking an hour to study the scheme just one time can save you countless hours of frustration later on.

**I/O ADDRESSES \$C000 THROUGH \$CFFF** All Apple systems, regardless of the amount of RAM installed in them, feature a section of addresses that refers to input/output (I/O) operations. Many of these addresses are not RAM or ROM locations in the usual sense; rather, they are memory-mapped I/O ports that can be addressed and otherwise treated as though they were true RAM or ROM locations.



This special portion of memory occupies address locations \$C000 to \$CFFF, or 49152 to 53247. Since the decimal versions of those addresses are greater than 32767, you can work with them from Integer BASIC by using a range of negative addresses between -16384 and -12289. (See Appendix A if you aren't sure about how and why to use negative addresses.)

Fig. 9-6 is an overall memory map of this rather unusual addressing range. The two lower and smaller sections are devoted to memory-mapped I/O functions; they are not RAM or ROM locations in the usual sense, but you can access them by referring to their addresses in an appropriate fashion.

The two upper sections are relatively large and are set aside for ROM operations from any cards that might be plugged into the peripheral card slots 1 through 7. If you aren't using any of these seven slots, there will be no memory devices in this range.

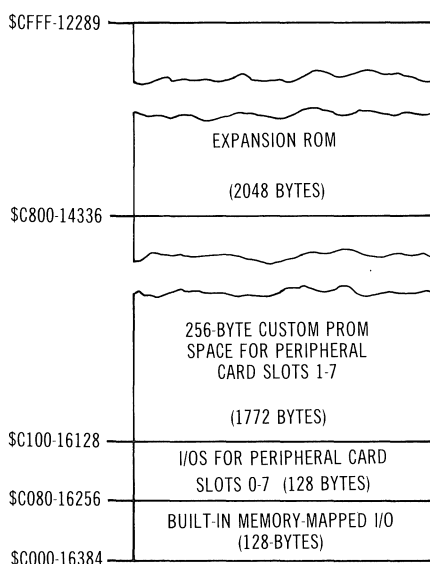


Fig. 9-6. I/O memory map.

**Built-In Memory-Mapped I/O: \$C000–\$C07F** This small, 128-byte section of addressing is devoted exclusively to built-in functions such as the game controls, loudspeaker, cassette tape IN and OUT, keyboard strobing, and the text screen mode “soft” switches. Much of the region is used by all Apple systems, regardless of the amount of RAM and extra peripheral devices. Fig. 9-7 maps this region in detail.

---

BEGINNING OF PERIPHERAL I/O		
\$C80(—16256)	GAME CONTROLLER STROBE	WRITE (POKE)
\$C070(—16272)	(16 IDENTICAL BYTES)	
\$C068(—16280)	SAME AS \$C060-\$C067	DON'T USE
\$C064(—16284)	ANALOG INPUTS GC0-GC3	READ (PEEK)
\$C060(—16288)	CASSETTE IN PUSHBUTTON INPUTS PB1-PB3	READ (PEEK)
\$C058(—16296)	ANNUNCIATOR "SOFT" SWITCHES ANO-AN3 (4 PAIRS OF BYTES)	WRITE (POKE)
\$C050(—16304)	SCREEN "SOFT" SWITCHES (4 PAIRS OF BYTES)	WRITE (POKE)
\$C040(—16320)	UTILITY STROBE OUTPUT (16 IDENTICAL BYTES)	READ (PEEK)
\$C030(—16336)	LOUDSPEAKER TOGGLE (16 IDENTICAL BYTES)	READ (PEEK)
\$C020(—16352)	CASSETTE OUT (16 IDENTICAL BYTES)	READ (PEEK)
\$C010(—16368)	CLEAR KEYBOARD STROBE (16 IDENTICAL BYTES)	(WRITE (POKE))
\$C000(—16384)	KEYBOARD STATUS INPUT (16 IDENTICAL BYTES)	READ (PEEK)

---

**Fig. 9-7. Built-in I/O function memory map.**

**Keyboard Status Input: \$C000–\$C00F**—All of these 15 keyboard status addresses contain the current status of the keyboard. They show whether or not a key has been depressed and the key code for the most recent keystroke. You can load any of those addresses to the A register from machine language, or assign it to a BASIC variable by doing a *PEEK addr*, where *addr* is the selected address. Most of the Apple literature, however, suggests using the lowest address in this range, \$C000 or -16384.

What kind of data can be found in that memory-mapped input area? The high-order “flag” bit in the data will be a 0 or a 1, depending on whether or not a keystroke has occurred. If a keystroke has occurred, that most-significant bit will be set to 1. Otherwise, it will be cleared to 0. In

terms of hexadecimal notation, that means the value fetched from address \$C000 will be \$80 or larger if a keystroke has occurred. Otherwise, it will be \$7F or less. From a decimal point of view, data PEEKed from -16384 will be 128, or greater, if the keystroke has occurred, but less than 128 if it has not.

What about the 7 lower-order bits? They carry the key code of the most recent keystroke whether the high-order “flag” bit is set or not.

*Clear Keyboard Strobe: \$C010–\$C01F*—Any keystroke sets the high-order “flag” bit in the keyboard status input address locations just described. To make the “flag” bit useful, there must be a provision for clearing it. You can clear it by writing to the clear keyboard strobe addresses.

POKEing or loading any sort of data to this range of addresses clears, or resets, the high-order keyboard input flag to 0. Most literature recommends loading a value of 0 to the lowest of these 15 addresses. Thus you can reset the keyboard strobe by doing a POKE -16368,0 from Integer BASIC, or by using this assembly language sequence:

```
LDA #$00
STA $C010
```

Failing to reset the keyboard strobe is tantamount to making a rapid and continuous series of keystrokes; the system “sees” a perpetual keystroke condition.

*Cassette OUT: \$C020–\$C02F*—This range of addresses memory-map directly to the cassette OUT jack on the rear of the Apple machine. Of those 15 locations, address \$C020, or -16352, is the one most recommended. Reading that address causes a single “click” at the cassette OUT jack; if you happen to have a cassette recorder running and set for recording, that “click” will be recorded for you. As you might imagine, doing a rapid series of read operations to that address causes a series of “clicks” that becomes an audio tone. The more rapidly a program executes those “clicks,” the higher the recorded frequency. With a bit of imagination, tinkering and time, you can write programs that record music onto cassette tape, using little more than PEEK (-16352) on LDA \$C020 instructions. The Apple system, of course, uses a similar technique for saving data and programs on tape—transforming entire blocks of 1s and 0s from memory into high and low audio tones.

*Loudspeaker Toggle: \$C030–\$C03F*—Read from any one of these 15 addresses, and you will get a single “click” from the loudspeaker. It is difficult to hear a single “click,” but a long series of them produces an

audio tone. The scheme is virtually identical to the cassette OUT operation.

You can, for instance, produce a wide range of audio tones by setting up a machine-language routine that does several series of LDA \$C030 instructions with different time intervals between instructions. The shorter the delay between instructions, the higher the audio tone will be.

It is possible to do the same sort of thing from Integer BASIC by executing a long series of PEEK(-16336) instructions. The only drawback here is that BASIC, being an interpretive high-level language, runs too slowly to produce the higher octaves of tones.

An interesting trick is to compose some tunes, using the loudspeaker port address to listen to, extend, or edit the music as desired. And when you are satisfied with it, change the memory-mapped I/O address reference from the loudspeaker to the cassette OUT jack (\$C020, or -16352). That allows you to record the music directly to cassette tape in such a fashion that it can be played from the tape without the aid of an Apple computer system.

*Utility Strobe Output: \$C040-\$C04F*—Fig. 9-8 shows the layout of the game controller socket that is located inside the Apple. This is the receptacle for the game-control paddles that are supplied with the system. Reading the utility strobe output addresses affects pin 5 of that socket.

Pin 5, the utility output pin, is normally at +5 V, but upon reading the utility strobe address \$C04F, the voltage level drops close to 0 V for about 0.5 microsecond. This function is used for strobing external, custom equipment in much the same fashion as you might strobe the loudspeaker or cassette OUT jack.

To generate that brief, negative-going pulse, simply do an LDA \$C04F from machine language or a PEEK(-16320) from BASIC.

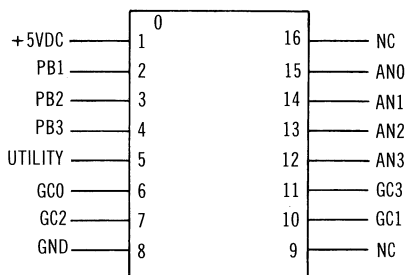


Fig. 9-8. Game controller socket.

*Set Graphics or Text: \$C050 and \$C051*—Writing to these two addresses sets up the video system for displaying either the text or graphics mode. These two “soft” switches behave in a mutually exclusive fashion; that is, writing to one of the “soft” switches activates it and automatically deactivates the other. For instance, writing to address \$C050, or -16304, sets the graphics mode and deactivates the text mode. Test this by trying a POKE -16304,0 from Integer BASIC or a sequence such as:

```
LDA #0
STA $C050
```

from machine language. On the other hand, writing to address \$C051 (-16303) sets the text mode and deactivates the graphics mode.

*Set Full or Mixed-Screen Text/Graphics: \$C052 and \$C053*—Writing a 0 to address \$C052 (-16302) sets up the video system for full-screen text or full-screen graphics. But writing a 0 to address \$C053 (-16301) restores the normal mixed text/graphics mode.

These are mutually exclusive operations; that is, writing to one of them automatically disables the other.

*Set Primary or Secondary Page: \$C054 and \$C055*—Writing a value of 0 to this set of “soft” switches sets up either the primary or secondary page of text and graphics. It doesn’t make much sense to attempt to view the primary and secondary pages of text or graphics at the same time, so it follows that these, too, are mutually exclusive switch settings.

POKEing or loading a zero to address \$C054 (-16300) sets up the system for displaying the primary page of whatever text or graphics scheme is active at the time. POKEing or loading a zero to address \$C055 (-16299) automatically deactivates the primary-page display and shows the secondary page.

*Set Text/Low-Resolution or High-Resolution: \$C056 and \$C057*—Writing to this set of mutually exclusive “soft” switches sets up either the text/ low-resolution graphics mode or the high-resolution graphics mode. Writing to \$C056 (-16298) sets up text/low-resolution graphics, while writing to its counterpart, address \$C057 (-16297), sets up the system for showing high-resolution graphics.

*Set or Clear AN0–AN3 Outputs: \$C058–\$C05F*—The diagram of the game controller socket in Fig. 9-8 shows four pins, labeled AN0, AN1, AN2, and AN3. You can set any one of these four pins to +5 V, or clear them to 0 V, in the same way you can set or clear the screen modes just described.

These “soft” switch connections are available for special external-device applications.

Table 9-6 shows that the functions are arranged in pairs of addresses. POKEing or loading a zero to the first in each pair clears the corresponding AN pin on the game controller to about 0 V. POKEing or loading a zero into the second address in each pair has the complementary effect: It sets the pin to about +5 V.

*Cassette IN Jack: \$C060*—Address \$C060 (-16288) is memory-mapped directly to the cassette IN jack on the rear of the Apple. Reading data from that address turns up a value of \$80 (128) or greater if the voltage present at that place is +1 V or more. It turns up a value of \$00 when the voltage is less than +1 V.

Of course, this is how the Apple monitor reads incoming data from cassette tape. A voltage at the cassette IN jack that is +1 V or more is interpreted as a logic-1 level, while a voltage that is less than +1 V is read as a logic-0 level.

*Pushbutton Inputs PB1–PB3: \$C061–\$C063*—Referring to Fig. 9-8, the diagram of the game controller socket, you will find three pin locations that are labeled PB1, PB2, and PB3. These are user-available versions of the cassette IN memory-mapped port just described. Reading from any one of these three addresses will yield a value of \$80 (128) or greater if the voltage at the corresponding game controller pin is +1 V or more. On the other hand, reading from the addresses will produce a value of \$00 (0) if the voltage at the pin is less than +1 V.

**Table 9-6. Set or Clear AN0–AN3 Outputs**

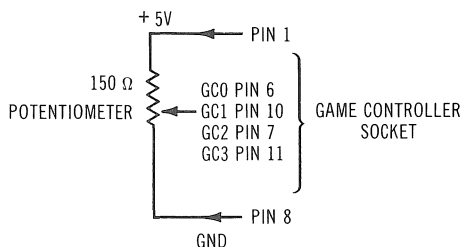
Hex Address	Decimal Address	Function
\$C058 \$C059	-16296 -16295	Clear AN0 (pin 15) to 0 V Set AN0 (pin 15) to 5 V
\$C05A \$C05B	-16294 -16293	Clear AN1 (pin 14) to 0 V Set AN1 (pin 14) to 5 V
\$C05C \$C05D	-16292 -16291	Clear AN2 (pin 13) to 0 V Set AN2 (pin 13) to 5 V
\$C05E \$C05F	-16290 -16289	Clear AN3 (pin 12) to 0 V Set AN3 (pin 12) to 5 V

There is some discrepancy in the literature regarding the labeling of these so-called pushbutton inputs. The labeling convention used in this book implies that the three pushbutton terminals on the game controller socket are simple extensions of the cassette IN connection. But if you are using the game paddles supplied with most Apple systems, you will find the pushbuttons on them labeled PB0 and PB1. Actually those pushbutton switches are plugged into pins 2 and 3 of the game controller socket, and ought to be labeled PB1 and PB2.

That need not create any real problems, however, as long as you refer to them with the appropriate addresses:

\$C061 (-16287) PB1 (pin 2)  
 \$C062 (-16286) PB2 (pin 3)  
 \$C063 (-16285) PB3 (pin 4)

*Analog Inputs: \$C064–\$C067*—These four input ports are available at the four game controller pins that are labeled GC0, GC1, GC2, and GC3. Once they are properly set (as described in connection with address \$C070 below), these terminals show a value of \$80 (128) or greater for a period of time that is directly proportional to the value of a potentiometer setting (see Fig. 9-9). The closer the potentiometer wiper arm is set to its +5 V connection, the longer it takes the analog value to drop below \$80. The maximum time is on the order of 3 milliseconds.



**Fig. 9-9. Game controller potentiometer.**

PEEK or load register A from the following addresses to pick up the status of the analog inputs:

\$C064 (-16284) GC0 (pin 6)  
 \$C065 (-16283) GC1 (pin 10)  
 \$C066 (-16282) GC2 (pin 7)  
 \$C067 (-16281) GC3 (pin 11)

*Analog Input Clear: \$C070*—You must read this address to start the timing operation for the four analog input ports just described. Writing to this address begins all four timing operations simultaneously.

**Peripheral Card I/O: \$C080–\$C0FF** This is a special memory-mapped I/O section of addresses that refers to functions included on any of the boards that might be inserted in the peripheral card slots. There are no standards for how these addresses may be used—it depends on how the cards' manufacturers want to use them. They can refer to memory locations on the cards or to I/O devices that are mapped to the addresses.

In any event, it is possible to read from or write to any one of 16 memory-mapped byte locations on each of the eight peripheral card slots. Table 9-7 summarizes all of those addresses.

**Table 9-7. Peripheral Card I/O Addresses**

Byte	Slot 0	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6	Slot 7
0	\$C080 -16256	\$C090 -16240	\$C0A0 -16224	\$C0B0 -16208	\$C0C0 -16192	\$C0D0 -16176	\$C0E0 -16160	\$C0F0 -16144
1	\$C081 -16255	\$C091 -16239	\$C0A1 -16223	\$C0B1 -16207	\$C0C1 -16191	\$C0D1 -16175	\$C0E1 -16159	\$C0F1 -16143
2	\$C082 -16254	\$C092 -16238	\$C0A2 -16222	\$C0B2 -16206	\$C0C2 -16190	\$C0D2 -16174	\$C0E2 -16158	\$C0F2 -16142
3	\$C083 -16253	\$C093 -16237	\$C0A3 -16221	\$C0B3 -16205	\$C0C3 -16189	\$C0D3 -16173	\$C0E3 -16157	\$C0F3 -16141
4	\$C084 -16252	\$C094 -16236	\$C0A4 -16220	\$C0B4 -16204	\$C0C4 -16188	\$C0D4 -16172	\$C0E4 -16156	\$C0F4 -16140
5	\$C085 -16251	\$C095 -16235	\$C0A5 -16219	\$C0B5 -16203	\$C0C5 -16187	\$C0D5 -16171	\$C0E5 -16155	\$C0F5 -16139
6	\$C086 -16250	\$C096 -16234	\$C0A6 -16218	\$C0B6 -16202	\$C0C6 -16186	\$C0D6 -16170	\$C0E6 -16154	\$C0F6 -16138
7	\$C087 -16249	\$C097 -16233	\$C0A7 -16217	\$C0B7 -16201	\$C0C7 -16185	\$C0D7 -16169	\$C0E7 -16153	\$C0F7 -16137
8	\$C088 -16248	\$C098 -16232	\$C0A8 -16216	\$C0B8 -16200	\$C0C8 -16184	\$C0D8 -16168	\$C0E8 -16152	\$C0F8 -16136



**Table 9-7—cont. Peripheral Card I/O Addresses**

Byte	Slot 0	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6	Slot 7
9	\$C089 -16247	\$C099 -16231	\$C0A9 -16215	\$C0B9 -16199	\$C0C9 -16183	\$C0D9 -16167	\$C0E9 -16151	\$C0F9 -16135
10	\$C08A -16246	\$C09A -16230	\$C0AA -16214	\$C0BA -16198	\$C0CA -16182	\$C0DA -16166	\$C0EA -16150	\$C0FA -16134
11	\$C08B -16245	\$C09B -16229	\$C0AB -16213	\$C0BB -16197	\$C0CB -16181	\$C0DB -16165	\$C0EB -16149	\$C0FB -16133
12	\$C08C -16244	\$C09C -16228	\$C0AC -16212	\$C0BC -16196	\$C0CC -16180	\$C0DC -16164	\$C0EC -16148	\$C0FC -16132
13	\$C08D -16243	\$C09D -16227	\$C0AD -16211	\$C0BD -16195	\$C0CD -16179	\$C0DD -16163	\$C0ED -16147	\$C0FD -16131
14	\$C08E -16242	\$C09E -16226	\$C0AE -16210	\$C0BE -16194	\$C0CE -16178	\$C0DE -16162	\$C0EE -16146	\$C0FE -16130
15	\$C08F -16241	\$C09F -16225	\$C0AF -16209	\$C0BF -16193	\$C0CF -16177	\$C0DF -16161	\$C0EF -16145	\$C0FF -16129

Suppose that a custom peripheral card is inserted into slot 5 and you want to write data \$80 to byte location 3 on that particular card. According to the table, you can access that device or memory location with address \$C0D3, or -16173. Writing a \$80 (128) to that location is a matter of doing:

```
LDA #128          LDA #$80
STA -16173        or     STA $C0D3
```

from a machine-language routine. Or working in Integer BASIC, you can accomplish the same thing with:

```
POKE -16173,128
```

**Peripheral Card ROM: \$C100–\$C7FF** Most commercially available peripheral cards for the Apple include some on-board ROM. This ROM contains built-in machine-language programming and data that are relevant to the function of the board. The Apple system sets aside some address locations especially for this purpose: 256 byte locations for cards 1 through 7. (Slot 0 is omitted from this function because it is dedicated to the same function in virtually every Apple.)

Table 9-8 maps the available ROM addressing range for each peripheral card slot. Reading from addresses in the range of \$C100 through \$C1FF (-16128 through -15873), for example, refers to addresses set aside for a 256-byte ROM on peripheral card number 1.

**Table 9-8. Peripheral Card ROM Addresses**

Slot	Hex Address	Decimal Address
1	\$C100-\$C1FF	-16128-15873
2	\$C200-\$C2FF	-15872-15617
3	\$C300-\$C3FF	-15616-15361
4	\$C400-\$C4FF	-15360-15105
5	\$C500-\$C5FF	-15104-14849
6	\$C600-\$C6FF	-14848-14593
7	\$C700-\$C7FF	-14592-14337

**Expansion ROM Space: \$C800-\$CFFF** If peripheral-card applications call for more than 256 bytes of ROM, you are free to use 2048 additional bytes located from address \$C800 through \$CFFF (-14336 through -12289). Reading from any address in this range automatically enables any ROM device located there. That can be a single, large 2K ROM or a series of smaller ones that are distributed among the peripheral cards.

**MAIN ROM ADDRESSES: \$D000 THROUGH \$FFFF** It is the fundamental nature of the 6502 microprocessor that makes it most feasible to place ROM at the very top of the addressing range. This area contains all of the built-in machine-language programming that makes the Apple perform as it does.

Fig. 9-10 maps this area for you, assuming that your system is using the Integer BASIC ROMs.

With the standard Apple ROMs for Integer BASIC, the lower portion of the map is dedicated to the special Programmer's Aid #1, which consists of:

HIGH-RESOLUTION GRAPHICS at \$D000 through \$D3FF (-12288 through -11265).

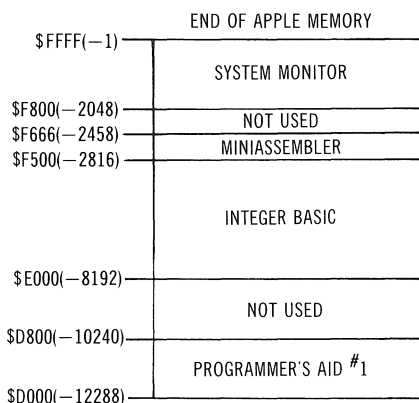
RENUMBER and APPEND at \$D400 through \$D4D4 (-11264 through -11060).

RELOCATE at \$D4DC through \$D5D2 (-11044 through -10798).

TAPE VERIFY at \$D535 through \$D5AA (-10955 through -10938).

RAM TEST at \$D5BC through \$D691 (-10820 through -10607).

MUSIC at \$D717 through \$D795 (-10473 through -10347).



**Fig. 9-10. ROM map.**

The ROM area from \$D800 to \$DFFF (-10240 through -8193) refers to ROM Socket D8 in the Apple hardware, but is largely unused by the Integer BASIC scheme and monitor. (ROM-based Applesoft uses this area, however.)

The machine-language programming for running Integer BASIC occupies ROM addresses \$E000 through \$F4FF (-8192 through -2817). The so-called “cold” entry point for Integer BASIC is at address \$E000 (-8192). This address is called by doing a CTRL-B keystroke. You can accomplish that same “cold” BASIC restart by doing a CALL -8192 or JSR \$E000.

The “warm” entry-point address for Integer BASIC is at address \$E003 (-8189). You can get to that point by doing a CTRL-C, CALL -8189, or JSR \$E003.

The miniassembler, described more fully in the next chapter, occupies ROM addresses \$F500 through \$F668 (-2816 through -2456). This feature is available only as long as you are using the Integer BASIC ROMs. In fact, it is this access to the miniassembler that has dictated the use of Integer BASIC (as opposed to Applesoft BASIC) throughout this book.

The system monitor, common to all Apple ROM configurations, takes up the very top portion of the addressing range: \$F800 through \$FFFF (-2048 through -1). Although the monitor programming begins at \$F800, its entry points are at higher addresses:

\$FF59 (-167)—“Cold” entry point, same as doing RESET.

\$FF65 (-155)—“Warm” entry point, usually used to get from BASIC to the monitor without destroying the values of any variables.



# Programming With the Mini assembler

The material in Chapter 8 included some 6502 machine-language instructions. As useful as those few instructions were, they hardly represent the full power of the 6502 and the Apple system. In this chapter, we will tell you the rest of the story. We will tell you how to prepare, load, test, debug, and execute a wide range of machine-language programs. The discussion features the Apple miniassembler—a good assembly-language programming aid that is built into the Integer BASIC ROMs. **10**

Recall from discussions in Chapter 8, if you will, that a complete machine-language listing includes both a source-code and an object-code version. The source-code, or assembly-language, version is written by hand in a semi-plain English form, using standard mnemonic expressions, labels, and comments. Its data and addresses may be expressed in either a decimal or hexadecimal form. It is a programming format that is tailored specifically to human understanding.

Once the programmer is satisfied with the assembly-language version, he or she has to employ some mechanism for translating the source-code instructions into a machine-compatible object-code version. Up until now, that has been done by hand by looking up the machine-language instructions for each of the assembly-language instructions. That is called a *hand assembly* process. As you might suspect, hand assembling a program can be a tedious, time consuming and error-prone job.

There is an alternative to hand assembly, however. All you need is a particular kind of utility computer program that does the translating job automatically. Such a program, called an *assembler*, accepts the humanly understandable assembly-language instructions and translates them into their machine-language counterparts.

The ROMs for Integer BASIC include a small, but quite useful, assembler routine. This so-called miniassembler lacks the most powerful features of some of the more sophisticated assemblers, but at least it elimi-

nates a lot of tedious work and lessens the chance of human error in translating from source code to object code.

When a programmer translates the original assembly-language program into machine language by hand or through an assembler, the next step in the procedure is to load the object-code version into the computer.

If you have assembled the program by hand for some reason, you can load the machine codes directly into RAM, one byte at a time, through the system monitor. But using the miniassembler is better because it will load the machine codes for you as you type in the source-codes from the keyboard. It is difficult to make a convincing case against using the miniassembler most of the time.

Once loaded into the system, the machine-language programs ought to be executed, tested, and debugged. The Apple *disassembler* feature is an invaluable aid in this respect. Being just the converse of the miniassembler, the disassembler reads machine-language instructions and translates them back into source-code mnemonics, for display on the screen or printing to a line printer. The miniassembler doesn't store your source-code listing, so being able to disassemble the machine-language version is an invaluable testing and debugging aid.

A working machine-language program ought to be saved on tape or disk for reloading at some later time. The Apple monitor handles such tasks quite well.

The overall purpose of this chapter is to deal with the procedures for preparing an assembly-language program, entering and assembling the program through the miniassembler, testing and debugging the program with the aid of the disassembler, and saving and reloading the program.

That is a lot of ground to cover in a single chapter, but we assume that you already have a basic understanding of the 6502 and its instruction set, and that you can handle hexadecimal expressions.

## A FIRST LOOK AT SOME ASSEMBLY-LANGUAGE PROGRAMMING

As mentioned earlier in this chapter, the Apple monitor includes a disassembler routine. It resides in the monitor ROMs that are common to all Apple systems and not in the Integer BASIC ROMs as the miniassembler does. The disassembler lets you explore any part of the Apple memory system, picking up blocks of machine-language instructions and translating them into source-code mnemonics on the screen. Try this:

1. Do a RESET to get the system running in the monitor mode. You know you are in the monitor when you see the asterisk as a prompt symbol.
2. Enter E000L from the keyboard. The screen should fill with 20 lines of assembly-language programming. (See Listing 10-1.)

## Listing 10-1. Disassembled ROM Listing.

---

```
E000— 20 00 F0 JSR    $F000
E003— 4C B3 E2 JMP    $E2B3
E006— 85 33   STA    $33
E008— 4C ED FD JMP    $FDED
E00B— 60     RTS
E00C— 8A     TXA
E00D— 29 20   AND    #$20
E00F— F0 23   BEQ    $E034
E011— A9 A0   LDA    #$A0
E013— 85 E4   STA    $E4
E015— 4C ED FD JMP    $FDED
E018— A9 20   LDA    #$20
E01A— C5 24   CMP    $24
E01C— B0 0C   BCS    $E02A
E01E— A9 8D   LDA    #$8D
E020— A0 07   LDY    #$07
E022— 20 ED FD JSR    $FDED
E025— A9 A0   LDA    #$A0
E027— 88     DEY
E028— D0 F8   BNE    $E022
```

---

Entering something such as E000L from the monitor instructs the system to disassemble 20 machine-language instructions, beginning at the designated address—\$E000 in this case. The general form of the disassembly command is

*addrL*

where *addr* is the hexadecimal starting address of the code to be disassembled, and the L suffix instructs the system to List the program in a standard assembly-language format. In this particular case, the disassembly begins at the beginning of the Integer BASIC ROM programming.

The left-hand column of four-character hexadecimal numbers that you see on the screen represents the address of the first machine-language code in each instruction. The first instruction in this example begins at address \$E000, the second begins at \$E003, the third at \$E006, and so on.

The machine-language version of the program appears in the second major column. They, too, are in a hexadecimal format (as opposed to the decimal versions shown in Chapter 8). Note that some are 1-byte codes, some are 2-byte codes, and several are 3-byte codes.

The right-hand columns show the 6502 mnemonics and their respective operands—the values of data or addresses that must be associated with many of them. (See the 6502 instruction set in the appendices for a brief interpretation of those mnemonics.)

You can, in principle at least, disassemble the entire Apple ROM system and discover in detail how it works and how you might use it to your

advantage. That is a lot of work, though, and it calls for a good understanding of 6502 programming.

For the disassembler to be truly useful, you must list the code from an address that holds the first code in an instruction. If you specify a listing from anywhere else, you will see a lot of the mnemonics replaced with question marks (???).

To see that idea at work, begin the disassembly in the middle of an instruction-code sequence by doing E001L from the monitor. That starts the disassembly from the second byte in the JSR \$F000 instruction shown in Listing 10-1. The result is a jumble of “nonsense” instructions and question marks in the first part of the listing on the screen. It does, however, manage to straighten itself out by the time it gets to address \$E066.

So whenever you are playing around with the disassembler and you see a lot of question marks appearing in the listings, one of two things is happening: Either you are beginning the disassembly in the middle of an otherwise valid instruction-code sequence, or you are attempting to disassemble a block of data.

To see the latter at work, do a 0L. That begins the disassembly at address \$0000—at the beginning of zero-page memory. That RAM space normally carries data bytes and 2-byte sequences that point to addresses elsewhere in the system. The disassembler cannot make sense of RAM space because it doesn’t contain machine-language instructions at all.

If you want to look at a sequence of disassembled instructions that is more than 20 instructions long, begin by entering the starting address followed by an L command. After observing the first 20 instructions, simply strike the L key again. That will cause the system to display the next 20 instructions. You can proceed in that fashion as long as you wish. And if you are using a line printer and want to print out a long series of disassembled instructions, do something such as this:

```
E000LLLLLLLL
```

That will print out eight consecutive groups of 20 disassembled instructions—a total of 160 instructions. Simply append the starting address with one L for each page of 20 instructions you want to print.

This disassembly technique is an especially powerful debugging tool for custom machine-language programs. Once you have entered a machine-language program, you can check its disassembled version at any later time by specifying a disassembly from the beginning of the program or at any other point that represents the first byte of a valid instruction.

**USING THE MINIASSEMBLER** The miniassembler is part of the Integer BASIC ROM set and you must have Integer BASIC in your system, even though you might not use BASIC programming at all.



**Entering the Mini-assembler** The entry-point address for the miniassembler is \$F666 (-2458). There are a couple of ways to reach that entry point, depending on whether you are using the monitor or Integer BASIC.

To enter the miniassembler from:  
The system monitor, enter F666G.  
Integer BASIC, enter CALL -2458.

(Whenever you are operating from the monitor, entering an address followed by a G character instructs the system to begin executing a routine from that address.)

As soon as you execute either of those commands, the system enters the miniassembler mode as indicated by an exclamation point (!) prompt symbol.

**Getting Out of the Mini-assembler** The procedure for getting out of the miniassembler depends on where you want to go from there. The logical choices are the system monitor, Integer BASIC, or any other routine residing in ROM or RAM.

To leave the miniassembler and go to:  
The system monitor, RESET or enter \$FF69G.  
Integer BASIC, enter \$E003G.  
Address *addr*, enter \$*addr*G.

So answering the exclamation point prompt symbol by striking the RESET key or entering a \$FF69G returns the system to the monitor command mode. Using \$FF69G is a bit more elegant than doing a brute-force RESET because the former does not clear a lot of system variables to their default values. And getting to Integer BASIC from the miniassembler by executing \$E003G also preserves any BASIC programming and variables you might have in the system at the time.

**Loading Programs Through the Mini-assembler** Once the miniassembler is up and running (as signalled by the exclamation point prompt symbol), you can specify an address for the first instruction, type that instruction, and continue typing source-code instructions until you are done. It is necessary to specify the address of the first instruction; the miniassembler takes care of all subsequent addressing.

Suppose that you have a short assembly-language routine that looks like this:

```
LDY #$FF ;INITIALIZE Y
LDX #$FF ;INITIALIZE X
LDA #$00 ;INITIALIZE A
RTS      ;RETURN TO CALLING ROUTINE
```

The only additional information required is the starting address of the routine. Let's make it \$0320.

Now, get into the miniassembler and respond to the exclamation point prompt symbol by typing the starting address, a semicolon, and the first instruction. Before you conclude that operation by striking the RETURN key, check to see if the screen looks like this:

```
!320:LDY #FF
```

Striking the RETURN key at this point causes the system to assemble the instruction and enter the machine-language codes into the appropriate address locations in RAM. The line you just typed is replaced with this:

```
0320-   A0 FF   LDY   #$FF
!
```

The prompt symbol indicates that the miniassembler is ready for the next instruction. For this point on, you no longer have to enter the address—just *type a space* followed by the next source-code instruction. Before you strike the RETURN key, the display should look something like this:

```
0320-   A0 FF   LDY   #$FF
! LDX #FF
```

And after striking the RETURN key:

```
0320-   A0 FF   LDY   #$FF
0322-   A2 FF   LDX   #$FF
```

Just don't forget to precede every instruction other than the first one with a space. Failing to type at least one space will bring up the miniassembler error symbol—a caret (^).

After entering the entire source-code program listed earlier, the screen should look like this:

```
0320-   A0 FF   LDY   #$FF
0322-   A2 FF   LDX   #$FF
```

```
0324-  A9 00    LDA    #$00
0326-  60      RTS
!
```

What you should do next depends on whether you want to enter further programming, try running the program just entered, save it on tape or disk, or quit for the day. Let's suppose that you want to run it.

**Running a Program from the Mini-assembler** Once you have entered a legitimate assembly-language program through the mini-assembler, you can run it by answering the exclamation point prompt symbol with a dollar sign, the desired starting address, and a G character.

Suppose that you want to run the program just described, beginning at address \$0320. This is what the display should look like:

```
!$320G
```

The system prints the exclamation point, but you type the dollar sign, address 320, and the G character. Notice that you must use the dollar sign ahead of the address and that there is no space between the prompt symbol and dollar sign. Omitting the dollar sign will bring up the mini-assembler caret-symbol error marker.

If you have been following these discussions by actually typing in the program, you will find that this particular program does nothing of any apparent use. It is important to notice, however, that the RTS instruction at the end of the program returns the system to the mini-assembler.

Incidentally, you can execute any self-contained machine-language routine from the mini-assembler mode. Try this command from the mini-assembler mode:

```
!$FC58
```

That executes a subroutine beginning from address \$FC58. That happens to be a monitor subroutine that homes the cursor and clears the screen. After that happens, the system returns to the mini-assembler with the prompt symbol appearing in the upper left-hand corner of the screen.

Running new programs from the mini-assembler is a good technique during the testing and debugging phases of the programming task.

**Running a Program From the Monitor** Of course it isn't absolutely necessary to run a program loaded under the mini-assembler mode from that same mode of operation. You can execute the program at a later time from the monitor.

Consider the short program described earlier. It begins at address \$0320, and loads \$FF to the X and Y registers and #00 to the A register before returning to the calling routine. Get out of the miniassembler and into the monitor mode by striking the RESET key or responding to the prompt symbol with \$FF69G.

Then, from the monitor, enter 320G. That will execute the little machine-language routine and return to the monitor.

Can that same routine be executed from Integer BASIC? Certainly it can. Get into Integer BASIC and do a CALL 800. That calls the routine by using the decimal version of the starting address. The system then executes the routine and returns to BASIC.

If you would like to work with a more convincing example, get back into the miniassembler mode and enter the program shown in Listing 10-2A. When you are done, the display should look like the one in Listing 10-2B.

### Listing 10-2. Loudspeaker Beeper.

---

LDY	#SOF	:SET FOR 15 BEEPS
JSR	\$FF3A	:BEEP THE LOUDSPEAKER
DEY		:DECREMENT THE COUNTER
BNE	\$0322	:IF NOT DONE, BEEP AGAIN
RTS		:ELSE RETURN TO CALLING ROUTINE

(A)

---

0320—	A0 0F	LDY	#SOF
0322—	20 3A FF	JSR	\$FF3A
0325—	88	DEY	
0326—	D0 FA	BNE	\$0322
0328—	60	RTS	

(B)

---

Try the program by doing a \$320G from the miniassembler. It should beep the loudspeaker 15 times and then return to the miniassembler.

Next, get into the system monitor mode and respond to the asterisk prompt symbol by entering 320G. The beeping routine will run again and return to the monitor mode.

Finally, get into Integer BASIC and execute a CALL 800 command. That, too, runs the beeping routine and returns to BASIC.

**Saving and Loading Tapes From Miniassembler** Saving and reloading machine-language programs is a vital part of any machine-language venture. It's a rather simple procedure as long as you can keep track of the first and last address used. The program in Listing 10-2B, for instance, uses RAM addresses \$0320 through \$0328, and you must keep a written record of that fact.

Saving that program on cassette tape from the miniassembler mode of operation is a matter of responding to the exclamation print prompt symbol with:

**\$320.328W**

Do not strike the RETURN key, however, until the cassette machine is running in the record mode. With the cassette running in the record mode, strike the RETURN key. The system will record that block of machine language and return to the miniassembler.

Of course you can accomplish the same feat from the monitor mode as well. It is only slightly easier because you do not have to prefix the address range with a dollar sign. The miniassembler requires that first dollar sign.

Getting a previously saved machine-language program back into the system calls for the same command, but appended with an R instead of a W. So the program just described can be loaded into the same address range by typing:

```
$320.328R
```

The cassette machine is then started in its play mode, and the RETURN key is struck. The system then loads the program and returns to the miniassembler.

**PREPARING ASSEMBLY-LANGUAGE PROGRAMS** The Apple miniassembler was specifically designed to use just a few bytes of RAM and enter machine language directly into specified addresses as the user types in the source-code program. Those are truly exciting features and a lot of the more sophisticated assemblers cannot make those claims.

But there are some serious trade-offs, too. Specifically, the Apple miniassembler cannot accept, or *support*, comments and labels. Comments, you recall, serve the same function as REM statements in BASIC; they provide a means for tacking on plain-English messages that explain what is going on at critical places in the program. Most assemblers do support comments, but the Apple accepts only assembly-language mnemonics and hexadecimal numbers. And labels, as you will soon see, provide a convenient and clear way to identify addresses that are to be used in an assembly-language program. The Apple miniassembler cannot accept labels.

The ROM listings in the back of the *Apple II Reference Manual* are a fine example of fully documented assembly-language programming. They were *not* done through the miniassembler, but on a full-blown, highly sophisticated assembler system. But what we do here by hand and with the miniassembler will work together to produce some documentation that is just as complete and useful.

**Using and Defining Labels** The first part of the ROM listings in your *Apple II Reference Manual* consists of a list of label definitions.

Labels LOC0 and LOC1, for instance, are defined as zero-page addresses \$00 and \$01, respectively. In the main body of the program, then, references to LOC0 and LOC1 actually refer to their addresses. Thus a source-code instruction such as LDA LOC1 is really identical to LDA \$01.

Using such labels in place of the addresses they represent makes it easier to write and read an assembly-language program. It is easier for most people to understand and remember what a label such as WNDLFT means than an address such as \$20. LDA \$20? What does that mean? LDA WNDLFT—ahh, yes. It means load the A register with the content of the WNDLFT address.

When preparing an assembly-language program, make generous use of labels. Use standard Apple labels wherever possible, but of course there will be many instances where you will have a chance to dream up some of your own. The important thing is to keep a running list of addresses that those labels represent.

Labels not only represent address locations of important bytes of data, but also critical entry points and special addresses within the program itself. Looking through the ROM listing, you will find a number of instructions such as JSR GBASCALC. A JSR instruction is supposed to be followed by a 2-byte address, but for the sake of convenience and some clarity, the programmers used the label GBASCALC instead. Look through the listing and you will find that GBASCALC is representing address \$F847.

Listing 10-3A shows a source-code listing that defines and uses labels. Labels SPKR and WAIT refer to addresses in the standard Apple system. SPKR EQU \$C030 means that label SPKR is identical to address \$C030—an address that represents the I/O port for the loudspeaker. And label WAIT refers to a monitor subroutine that executes a time delay propor-

**Listing 10-3. Demonstration of Labels.**

ORG	EQU	\$0320					
SPKR	EQU	\$C030					
WAIT	EQU	\$FCA8					
START	LDY	#FF	:INITIALIZE LONG DELAY				
CLICK	LDA	SPKR	:CLOCK THE LOUDSPEAKER				
	TYA		:GET DELAY FROM Y				
	JSR	WAIT	:DO A TIME DELAY				
	DEY		:DECREMENT DELAY IN Y				
	BNE	CLICK	:IF NOT DONE, CLICK AGAIN	0320—	A0 FF	LDY	\$FFF
	BEQ	START	:OTHERWISE START OVER	0322—	AD 30 C0	LDA	\$C030
				0325—	98	TYA	
				0326—	20 A8 FC	JSR	\$FCA8
				0329—	88	DEY	
START	EQU	—		032A—	D0 F6	BNE	\$0322
CLICK	EQU	—		032C—	F0 F2	BEQ	\$0320

(A)

(B)

tional to the value of a number in the A register. It is defined by `WAIT EQU $FCA8`.

`START` and `CLICK`, on the other hand, are home-brewed labels that refer to addresses within the program itself. `START EQU $0320` means that address `START` is at `$0320`; it marks the address of the first instruction in the program. Use any other label you like, but I think that `START` is quite appropriate. Then there is the `CLICK` label. `CLICK` refers to the address of the `LDA SPKR` instruction, and the `BNE` instruction near the end of the program uses it.

Labels indicating critical points in an assembly-language program are especially important because the programmer does not have to keep track of the actual addresses. Let the miniassembler do that for you later on.

That source-code listing is what you should have at hand when you approach the miniassembler. After loading the program, you should be able to list it as shown in Listing 10-3B. Between those two listings—your source-code version and the disassembled version—your documentation is as complete as it ever has to be.

**Loading Through the Miniassembler** Given a complete source-code program listing, the next step is to enter the program into the Apple system via the miniassembler. We are going to show how to load the program in Listing 10-3A. Follow along carefully, because it demonstrates how to deal with the labels.

With Listing 10-3A at hand, get into the miniassembler and type in the `ORG` address, `$320`, and the first instruction. Just prior to striking the `RETURN` key, the display looks something like this:

```
!320:LDY #FF
```

After striking the `RETURN` key, the presentation changes to this:

```
0320-    A0 FF    LDY    #$FF
!
```

That first address, `$0320`, also represents the address for the `START` label. Make a note of that fact on your source-code listing.

The next instruction refers to label `SPKR`, but the miniassembler won't accept the label—it needs the address it represents. So looking back to your source-code listing to get that address, type in the instruction as:

```
! LDA C030
```

After striking the `RETURN` key, the screen should show this sequence:

```

0320-  A0 FF    LDY    #$FF
0322-  AD 30 C0 LDA    $C030
!
```

Address \$0322 marks the special CLICK label in the source-code listing. Make a note of that on the listing. You will need it later.

Enter the TYA instruction and then the JSR instruction, using \$FCA8 in place of the WAIT label. Things are fairly straightforward for the last two instructions. Enter the BNE instruction using \$0322 in place of CLICK and \$0320 in place of START.

When the job is done, the listing should appear as in Listing 10-3B.

Since the miniassembler does not support labels, you must define the ones you know in advance and keep track of those you discover as you go along.

This particular example lets you see the address for label CLICK before you need it later in the program. There are some instances where you need a label address *before* you get far enough along in the program to know what it will be. One trick for handling that situation is to use any old address—perhaps something quite distinctive such as \$0000—when you need it. Then after you find out what that address really is, you can edit the listing to insert the proper address.

As an exercise in interpreting labels, see if you can load the source-code program in Listing 10-4. Complete the documentation by indicating the addresses for labels START and CLICK.

#### Listing 10-4. Exercise in Use of Labels.

---

```

ORG      EQU    $0320
HOME     EQU    $FC58
SPKR     EQU    $C030
COUT1    EQU    $FDF0
WAIT     EQU    $FCA8

START    JSR     HOME      :HOME CURSOR AND CLEAR SCREEN
        LDY     #$FF      :SET INITIAL DELAY IN Y
CLICK    LDA     SPKR      :CLOCK THE SPEAKER
        LDA     #$1A      :LOAD INVERSE Z CHARACTER
        JSR     COUT1     :PRINT IT
        TYA                      :GET CURRENT DELAY FROM Y
        JSR     WAIT      :DO A TIME DELAY
        DEY                      :DECREMENT DELAY IN Y
        BNE     CLICK     :IF NOT DONE, CLICK AGAIN
        BEQ     START     :OTHERWISE, START OVER

START    EQU     _____
CLICK    EQU     _____
```

---



**LOADING THROUGH THE MONITOR** If you must work in machine language, it is more convenient to load through the monitor. Loading hexadecimal bytes through the monitor is a simple matter of specifying the starting address of the sequence of bytes, typing a colon, and then typing in the bytes, each separated by a space. When Listing 10-3 is loaded in this fashion, the presentation on the screen looks something like this:

```
*320:A0 FF AD 30 C0 98 20 A8 FC 88 D0 F6 F0 F2
```

That loads the machine-language program, beginning at address \$032; striking the RETURN key signals the end of the operation. After that, you can see the disassembled version of the program by executing a 320L from the monitor. It will look just like the version in Listing 10-3B. Running the program is then a matter of executing a 320G from the monitor.

Most programmers agree that it is easier and faster to type in machine language through the monitor than through the miniassembler as long as you must use machine language. One area in which you should use machine language is data tables. Data tables have no mnemonics; they are strictly numerical. That means you are better off loading data tables as hexadecimal bytes—an operation better performed through the monitor.

Finally, the monitor offers the most convenient means for altering a single byte or two within an existing machine-language program. Perhaps it is desirable to change an existing LDA #\$FF statement to LDA #\$0F. Rather than getting into the miniassembler and typing the entire instruction, determine the address of the byte to be changed and write over it by entering the revised version through the monitor.

The disassembled version of the instruction will look like this:

```
E011-    A9 FF      LDA #$FF
```

To change the #\$FF to #\$0F from the monitor, do this:

```
E012:FF
```

and strike the RETURN key.

The FF byte, you see, is located at address E011—the byte immediately following the E011 address shown in the disassembled version of the program. After making that change, the disassembled version will look like this:

```
E011-    A9 0F      LDA #$0F
```

**DEBUGGING WITH THE BRK INSTRUCTION**      Machine-language programs generally run very fast—far too fast to follow in detail. About all you can know for sure is the status of the system at the beginning and the end of the execution of such a program. Keeping continuous track of events taking place during the program is an impossible task. But keeping track of events is important for debugging purposes.

The BRK instruction offers some hope for keeping track of events through the execution of a machine-language program. Insert a temporary BRK instruction at the beginning of any other instruction in the program, and things come to a halt as soon as the system executes the BRK. And the Apple is set up so that it automatically displays the contents of the internal registers at the conclusion of a BRK operation.

So let's suppose that you are working with a machine-language program such as the one in Listing 10-3, and you find it isn't working right. Disassemble it to find the address of some critical instruction, then load a 00 (the machine code for BRK) at that address, through the monitor. Run the program, and it stops at the BRK instruction and prints out the contents of the registers. You can also examine other memory locations through the monitor while the system is stopped. Then when you want to get the program running normally again, simply replace the BRK instruction with the original one.

Two additional debugging tools, STEP and TRACE, amount to a powerful family of troubleshooting aids. Both of these are well documented in the standard Apple reference manuals.

# Appendix

## Number-System Base Conversions

Just about any computer (certainly the Apple II) is essentially a **A** binary machine; the 6502 microprocessor does all of its control, arithmetic, and logic operations in a base 2, or *binary*, number system. And it so happens that the 6502 works with 8-bit binary numbers—a full *byte* of them.

People do not think and work with binary numbers very well, however. Such numbers, being made up exclusively of 0s and 1s, are very cumbersome. One alternative to purely binary representations of numbers is *hexadecimal* numbers. The hexadecimal (base 16) number system looks at binary numbers in groups of four; every group of four binary numbers (sometimes called a *nibble*) can be represented by a single hexadecimal number. So, instead of having to work with strings of eight 0s and 1s in base 2 binary, it is possible to work with just two hexadecimal characters.

While, indeed, many machine-language programmers can learn to work with hexadecimal numbers with great proficiency, the general population still prefers the ordinary decimal (base 10) number system. Apple engineers were aware of that fact, and Integer and Applesoft BASIC are built around the decimal number system exclusively.

As long as one works with Apple BASIC in its most elementary fashion—doing no special addressing or machine-language work—there is no need to be aware of hexadecimal or binary numbers. But hexadecimal numbers become quite helpful when doing extensive machine-language programming.

Thus, programmers who are working their way deeper and deeper into the Apple II system will find themselves having to make conversions between decimal and hexadecimal numbers and, eventually, between binary and hexadecimal numbers. The purpose of this appendix is to make such conversion tasks as simple as possible.

There are many ways to approach the conversions between these three different number systems; the following are the most straightforward.

**HEXADECIMAL-TO-DECIMAL CONVERSIONS** In the Apple II system, data is carried as a 1-byte (two-hexadecimal-number) code. Addresses are carried as 1-byte codes for the zero-page memory and as 2-byte codes for the remainder of the usable memory space. Table A-1 can be very helpful for translating hexadecimal numbers into their decimal counterparts. This sort of situation often arises when one is writing programs in both BASIC and machine language.

The table can be used for converting up to four hexadecimal places, or nibbles, to their decimal counterpart. Notice that there are four major columns, labeled 1 through 4. These column numbers represent the relative positions of the hexadecimal characters as they are usually written, with the least-significant nibble on the right and the most-significant nibble on the left.

**Table A-1. Hexadecimal/Decimal Conversion**

MSB 4		3		2		LSB 1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4096	1	256	1	16	1	1
2	8192	2	512	2	32	2	2
3	12288	3	768	3	48	3	3
4	16384	4	1024	4	64	4	4
5	20480	5	1280	5	80	5	5
6	24576	6	1536	6	96	6	6
7	28672	7	1792	7	112	7	7
8	32768	8	2048	8	128	8	8
9	36864	9	2304	9	144	9	9
A	40960	A	2560	A	160	A	10
B	45056	B	2816	B	176	B	11
C	49152	C	3072	C	192	C	12
D	53248	D	3328	D	208	D	13
E	57344	E	3584	E	224	E	14
F	61440	F	3840	F	240	F	15

To see how the table works, suppose that you want to convert the hexadecimal value \$1A3F into decimal. The first character on the left takes a decimal equivalent shown in column 4—4096. The second character from the left takes on the value from column 3—2560. The last two figures get their decimal equivalents from columns 2 and 1—48 and 15, respectively. Then, to get the true decimal value, add those decimal equivalents:  $4096+2560+48+15$ , or 6719. In other words, \$1A3F is equal to 6719 in decimal.

If you are converting a two-place hexadecimal number, just use columns 2 and 1. Hexadecimal \$C3, for instance, is equal to  $192+3$ , or decimal 195.

Table A-1 is adequate for hexadecimal-to-decimal conversions for all the usual sort of work on the Apple.

**DECIMAL-TO-HEXADECIMAL CONVERSIONS** When working back and forth between BASIC and machine-language routines, it is often necessary to convert decimal data and addresses into hexadecimal notation. Table A-1 comes to the rescue again. The procedure is a rather straightforward one, but it involves several steps.

Suppose, for example, that you want to convert decimal 65 into its hexadecimal counterpart. First, find the decimal number on the table that is equal to, or less than, the desired decimal number. The decimal number in this example is 65, and the closest value less than 65 is 64. The 64 is equivalent to a hexadecimal \$4 in column 2. Thus the most-significant number in the hexadecimal representation is 4.

Next, subtract that 64 from the number that you are working with:  $65-64=1$ . Then look up the hexadecimal value of the 1 in the next-lower column of the table—column 1 in this instance. The hexadecimal version of that number is \$1. Putting together those two hexadecimal characters, you get a \$41. Indeed, decimal 65 translates into hexadecimal \$41.

By way of a somewhat more involved conversion, suppose that you must convert decimal 19314 into hexadecimal notation.

Looking through the columns of decimal numbers in the table, you find that 16384 is the next-lower value; it translates into hexadecimal \$4 in column 4. So you are going to end up with a four-digit hexadecimal number, with the digit on the left being a 4.

To get the next-lower place value, subtract the table value 16384 from 19314:  $19314-16384=2930$ . The next-lower decimal value in this case is 2816 from column 3; that turns up a \$B as the next hexadecimal character. So far, the number is \$4B.

Now subtract the table value 2816 from 2930:  $2930-2816=114$ . The next-lower decimal value from column 2 is 112, and its hexadecimal counterpart is 7. And to this point, the hexadecimal number is \$4B7.

Finally, subtract the table value 112 from 114:  $114 - 112 = 2$ . From column 1, decimal 2 is the same as hexadecimal \$2; so the final hexadecimal character is \$2.

Putting this all together, it turns out that decimal 19314 is the same as hexadecimal \$4B72. Fig. A-1 summarizes the operation.

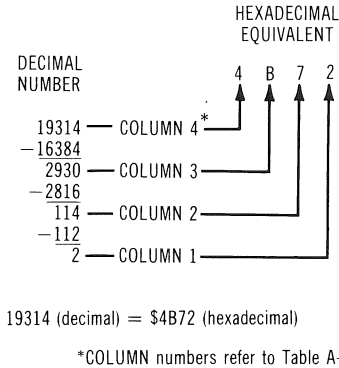


Fig. A-1. Converting decimal to hexadecimal.

**CONVENTIONAL DECIMAL TO 2-BYTE DECIMAL FORMAT** When POKEing addresses as 2-byte numbers into memory, it is necessary to convert the address to be affected into a 2-byte format. In decimal, such an operation isn't easy, but it is all a part of setting up address locations in decimal-oriented BASIC.

By way of an example, suppose that you are to load a 2-byte version of decimal address 1234 into memory addresses 16787 and 16788. That number to be stored, 1234, is too large for either of those 1-byte addresses, so it has to be broken up into two parts: one for each of the address locations.

Before a decimal number can be divided into a 2-byte version, it must be converted into hexadecimal form. Using the decimal-to-hexadecimal conversion described in the previous section, you find that decimal 1234 is equal to hexadecimal \$04D2.

Next, you divide that hexadecimal version of the number into two bytes: the most-significant byte (MSB) is \$04, and the least-significant byte (LSB) is \$D2. Divided that way, you end up with two 1-byte hexadecimal values: \$04 and \$D2.

Finally, convert those two sets of hexadecimal numbers into their decimal equivalents, treating them as two separate hexadecimal values. Thus \$04 converts to decimal 4, and \$D2 converts to 210.

The 2-byte version of decimal 1234 is thus 4 and 210, with 4 being the MSB and 210 being the LSB.

That takes care of the conversion of an ordinary decimal number into a 2-byte version, also in decimal. Now you must POKE these numbers into decimal addresses 16787 and 16788.

If you place the LSB of the 2-byte number into the lower-numbered address, the BASIC operation for satisfying the requirements of the example looks like this:

```
POKE 16787,210 : POKE 16788,4
```

No, it isn't a simple procedure to convert an ordinary decimal number into a 2-byte decimal format, but it's the price that must be paid for working with a byte-oriented machine in a decimal-oriented BASIC language.

## TWO-BYTE DECIMAL TO CONVENTIONAL DECIMAL FORMAT

Suppose that you are analyzing a machine-language routine that is presented in a decimal-oriented, BASIC format. Under that condition, a 2-byte address appears as a set of two decimal numbers; if you want to get that pair of numbers into a conventional decimal format, you have to play with them a bit.

Consider an instance where 223 turns up as the LSB in decimal, and 104 is the MSB. What address, or 2-byte decimal number, do they represent?

First, convert both sets of numbers into their hexadecimal counterparts: decimal 223=\$DF, and decimal 104=\$68. Since \$DF is the MSB and \$68 is the MSB, the overall hexadecimal representation of that 2-byte decimal format is \$DF68.

All that remains to be done is to convert that hexadecimal number into its full decimal counterpart: \$DF68=24567+2048+208+15=26849. That's it—the conventional representation of the original 2-byte decimal values. The combination of decimal numbers 223 and 104 actually points to decimal 26849.

## CONVERTING LARGE DECIMAL VALUES TO SMALLER NEGATIVE VALUES

Integer BASIC allows numbers having values from -32767 to 32767. Try working with a number outside that range, and you immediately get a 32767 ERR message. The possible range of Apple II addressing is different; it runs from 0 to 65535—all positive values. So there is bound to be some difficulty when attempting to CALL a machine-language routine that begins at memory addresses above 32767 (and there are a good many valuable monitor routines between 32768 and 65535).

The way around the problem is to CALL a negative address. It is a simple procedure as long as you can remember a “magic” number: 65536. Whenever you want to CALL an address that is greater than 32767, simply subtract the address from 65536 and stick a minus sign in front of it. That’s all there is to it.

Suppose that you want to call a routine that is located at address 40668. You can’t do a CALL 40668 without getting 32767 ERR from Integer BASIC. So subtract 40668 from 65536, and put a minus sign in front of it: -24868. Then you can get to that routine by doing a CALL -24868.

**CONVERTING NEGATIVE DECIMAL VALUES TO LARGER POSITIVE VALUES** Anyone who has used Integer BASIC knows that CALL -936 homes the cursor and clears the screen. That CALL refers to the starting address of a home-and-clear routine in the Apple II monitor. But what is the actual decimal address?

The conversion is a simple one if you use the “magic” number, 65536. In this case,  $65536 - 936 = 64600$ . The actual address of -936 is 64600. Just add the negative address to 65536.

**BINARY-TO-DECIMAL CONVERSION** In practice, most binary-to-decimal conversions are carried out with 1-byte (or 8-bit) binary numbers, although there are occasions when it is necessary to do the conversion from 2-byte (16-bit) numbers.

Fig. A-2 shows the breakdown of an 8-bit binary number. The positions are labeled 0 through 7, with zero indicating the least-significant bit position. Each of those 8-bit locations contains either a 0 or a 1.

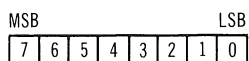


Fig. A-2. Eight-bit binary number.

Suppose that you want to POKE 01101011. But you have to use a decimal version of that binary number from BASIC. Here is how to go about determining that decimal version.

First, multiply the 0 or 1 in each bit location by  $2^n$ , where  $n$  is the bit place value in each case. Then simply add the results. (See the example in Fig. A-3.)

The same idea applies to converting 16-bit binary to a decimal equivalent. The place values run from 0 to 15 in that case, and Table A-2 can help you determine those larger powers of 2.



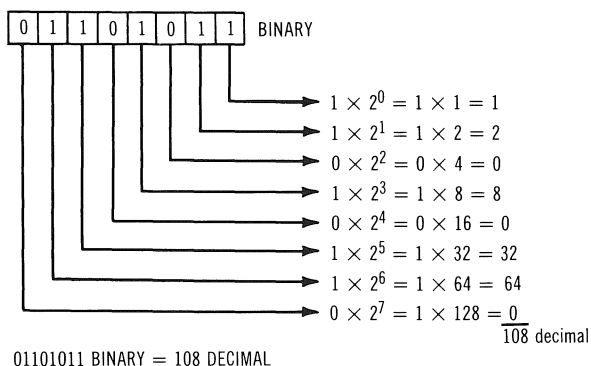


Fig. A-3. Converting binary to decimal.

Table A-2. Powers of Two

n	2 <sup>n</sup>
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1 024
11	2 048
12	4 096
13	8 192
14	16 384
15	32 768

Table A-3. Binary/Hexadecimal Conversion

Binary	Hexadecimal
0000	\$0
0001	\$1
0010	\$2
0011	\$3
0100	\$4
0101	\$5
0110	\$6
0111	\$7
1000	\$8
1001	\$9
1010	\$A
1011	\$B
1100	\$C
1101	\$D
1110	\$E
1111	\$F

**BINARY-TO-HEXADECIMAL CONVERSION** Converting a binary number into a hexadecimal format is perhaps the simplest of all the conversion operations. All you have to do is group the binary number into sets of 4 bits each, beginning with the least-significant bit, and then find the hexadecimal value for each group. Table A-3 helps with the latter operation.

Suppose the binary number is 10011101. There are two sets of 4 bits (or nibbles) here, 1001 and 1101. The hexadecimal equivalent is 9 for the first set, and D for the second set, as Table A-3 shows. Therefore, the hexadecimal version of that 8-bit binary number is \$9D.

The same procedure works equally well for 16-bit numbers; the only difference is that you end up with four hexadecimal characters instead of just two.

**HEXADECIMAL-TO-BINARY CONVERSION** Converting a hexadecimal number to its binary form is a simple matter of applying Table A-3 to change each hexadecimal character into the appropriate groups of 4 binary bits.

Example: Convert address \$404D into a binary format. According to the table, that hexadecimal number can be represented as 0100 0000 0011 1101.

**DECIMAL-TO-BINARY CONVERSION** There are several commonly cited algorithms for mathematically converting any decimal number into its binary format. But it is simpler in the long run, and probably more accurate, to use a two-step procedure.

The general idea is to convert the decimal number into its hexadecimal counterpart as described earlier in this appendix. Then convert the hexadecimal characters into their binary versions as described in the previous section.

Example: Convert 1234 decimal into binary. First, as described earlier, calculate the hexadecimal version of decimal 1234. Your answer should come out to be \$04D2. And that hexadecimal number, expressed in binary (from Table A-3) is 0000 0100 1101 0010. Thus 1234 is equal to binary 10011010010. You may include the five leading zeros if you wish.

# Appendix

## Character Codes for Text Printing Operations

The following tables show the hexadecimal (Hex) and decimal (Dec) codes that can be loaded or POKEd to video memory to print the designated character (Char).

**B**

**Table B-1. Inverse Screen Text Characters**

Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
\$00	(0)	@	\$10	(16)	P	\$20	(32)		\$30	(48)	0
\$01	(1)	A	\$11	(17)	Q	\$21	(33)	!	\$31	(49)	1
\$02	(2)	B	\$12	(18)	R	\$22	(34)	"	\$32	(50)	2
\$03	(3)	C	\$13	(19)	S	\$23	(35)	#	\$33	(51)	3
\$04	(4)	D	\$14	(20)	T	\$24	(36)	\$	\$34	(52)	4
\$05	(5)	E	\$15	(21)	U	\$25	(37)	%	\$35	(53)	5
\$06	(6)	F	\$16	(22)	V	\$26	(38)	&	\$36	(54)	6
\$07	(7)	G	\$17	(23)	W	\$27	(39)	'	\$37	(55)	7
\$08	(8)	H	\$18	(24)	X	\$28	(40)	(	\$38	(56)	8
\$09	(9)	I	\$19	(25)	Y	\$29	(41)	)	\$39	(57)	9
\$0A	(10)	J	\$1A	(26)	Z	\$2A	(42)	*	\$3A	(58)	:
\$0B	(11)	K	\$1B	(27)	[	\$2B	(43)	+	\$3B	(59)	;
\$0C	(12)	L	\$1C	(28)	\	\$2C	(44)	,	\$3C	(60)	<
\$0D	(13)	M	\$1D	(29)	]	\$2D	(45)	-	\$3D	(61)	=
\$0E	(14)	N	\$1E	(30)	^	\$2E	(46)	.	\$3E	(62)	>
\$0F	(15)	O	\$1F	(31)	—	\$2F	(47)	/	\$3F	(63)	?

**Table B-2. Flashing Screen Text Characters**

Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
\$40	(64)	@	\$50	(80)	P	\$60	(96)		\$70	(112)	0
\$41	(65)	A	\$51	(81)	Q	\$61	(97)	!	\$71	(113)	1
\$42	(66)	B	\$52	(82)	R	\$62	(98)	"	\$72	(114)	2
\$43	(67)	C	\$53	(83)	S	\$63	(99)	#	\$73	(115)	3
\$44	(68)	D	\$54	(84)	T	\$64	(100)	\$	\$74	(116)	4
\$45	(69)	E	\$55	(85)	U	\$65	(101)	%	\$75	(117)	5
\$46	(70)	F	\$56	(86)	V	\$66	(102)	&	\$76	(118)	6
\$47	(71)	G	\$57	(87)	W	\$67	(103)	'	\$77	(119)	7
\$48	(72)	H	\$58	(88)	X	\$68	(104)	(	\$78	(120)	8
\$49	(73)	I	\$59	(89)	Y	\$69	(105)	)	\$79	(121)	9
\$4A	(74)	J	\$5A	(90)	Z	\$6A	(106)	*	\$7A	(122)	:
\$4B	(75)	K	\$5B	(91)	[	\$6B	(107)	+	\$7B	(123)	;
\$4C	(76)	L	\$5C	(92)	\	\$6C	(108)	,	\$7C	(124)	<
\$4D	(77)	M	\$5D	(93)	]	\$6D	(109)	-	\$7D	(125)	=
\$4E	(78)	N	\$5E	(94)	^	\$6E	(110)	.	\$7E	(126)	>
\$4F	(79)	O	\$5F	(95)	—	\$6F	(111)	/	\$7F	(127)	?

**Table B-3. NORMAL-1 Screen Text Characters**

Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
\$80	(128)	@	\$90	(144)	P	\$A0	(160)		\$B0	(176)	0
\$81	(129)	A	\$91	(145)	Q	\$A1	(161)	!	\$B1	(177)	1
\$82	(130)	B	\$92	(146)	R	\$A2	(162)	"	\$B2	(178)	2
\$83	(131)	C	\$93	(147)	S	\$A3	(163)	#	\$B3	(179)	3
\$84	(132)	D	\$94	(148)	T	\$A4	(164)	\$	\$B4	(180)	4
\$85	(133)	E	\$95	(149)	U	\$A5	(165)	%	\$B5	(181)	5
\$86	(134)	F	\$96	(150)	V	\$A6	(166)	&	\$B6	(182)	6
\$87	(135)	G	\$97	(151)	W	\$A7	(167)	'	\$B7	(183)	7
\$88	(136)	H	\$98	(152)	X	\$A8	(168)	(	\$B8	(184)	8
\$89	(137)	I	\$99	(153)	Y	\$A9	(169)	)	\$B9	(185)	9
\$8A	(138)	J	\$9A	(154)	Z	\$AA	(170)	*	\$BA	(186)	:
\$8B	(139)	K	\$9B	(155)	[	\$AB	(171)	+	\$BB	(187)	;
\$8C	(140)	L	\$9C	(156)	\	\$AC	(172)	,	\$BC	(188)	<
\$8D	(141)	M	\$9D	(157)	]	\$AD	(173)	-	\$BD	(189)	=
\$8E	(142)	N	\$9E	(158)	^	\$AE	(174)	.	\$BE	(190)	>
\$8F	(143)	O	\$9F	(159)	—	\$AF	(175)	/	\$BF	(191)	?

**Table B-4. NORMAL-2 Screen Text Characters**

Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
\$C0	(192)	@	\$D0	(208)	P	\$E0	(224)		\$F0	(240)	0
\$C1	(193)	A	\$D1	(209)	Q	\$E1	(225)	!	\$F1	(241)	1
\$C2	(194)	B	\$D2	(210)	R	\$E2	(226)	''	\$F2	(242)	2
\$C3	(195)	C	\$D3	(211)	S	\$E3	(227)	#	\$F3	(243)	3
\$C4	(196)	D	\$D4	(212)	T	\$E4	(228)	\$	\$F4	(244)	4
\$C5	(197)	E	\$D5	(213)	U	\$E5	(229)	%	\$F5	(245)	5
\$C6	(198)	F	\$D6	(214)	V	\$E6	(230)	&	\$F6	(246)	6
\$C7	(199)	G	\$D7	(215)	W	\$E7	(231)	'	\$F7	(247)	7
\$C8	(200)	H	\$D8	(216)	X	\$E8	(232)	(	\$F8	(248)	8
\$C9	(201)	I	\$D9	(217)	Y	\$E9	(233)	)	\$F9	(249)	9
\$CA	(202)	J	\$DA	(218)	Z	\$EA	(234)	*	\$FA	(250)	:
\$CB	(203)	K	\$DB	(219)	[	\$EB	(235)	+	\$FB	(251)	;
\$CC	(204)	L	\$DC	(220)	\	\$EC	(236)	,	\$FC	(252)	<
\$CD	(205)	M	\$DD	(221)	]	\$ED	(237)	-	\$FD	(253)	=
\$CE	(206)	N	\$DE	(222)	^	\$EE	(238)	.	\$FE	(254)	>
\$CF	(207)	O	\$DF	(223)	-	\$EF	(239)	/	\$FF	(255)	?

# Appendix

## Organization of the Text/Low-Resolution Graphics Video Memory

These tables show the range of hexadecimal and decimal addresses for each of the 24 lines of text and low-resolution graphics.

C

**Table C-1. Primary-Page Memory Addresses**

Line	Hex Range	Dec Range
0	\$0400–\$0427	1024–1063
1	\$0480–\$04A7	1152–1191
2	\$0500–\$0527	1280–1319
3	\$0580–\$05A7	1408–1447
4	\$0600–\$0627	1536–1575
5	\$0680–\$06A7	1664–1703
6	\$0700–\$0727	1792–1831
7	\$0780–\$07A7	1920–1959
8	\$0428–\$044F	1064–1103
9	\$04A8–\$04CF	1192–1231
10	\$0528–\$054F	1320–1359
11	\$05A8–\$05CF	1448–1487
12	\$0628–\$064F	1576–1615
13	\$06A8–\$06CF	1704–1743
14	\$0728–\$074F	1832–1871
15	\$07A8–\$07CF	1960–1999

**Table C-1—cont. Primary-Page Memory Addresses**

16	\$0450–\$0477	1104–1143
17	\$04D0–\$04F7	1232–1271
18	\$0550–\$0577	1360–1399
19	\$05D0–\$05F7	1488–1527
20	\$0650–\$0677	1616–1655
21	\$06D0–\$06F7	1744–1783
22	\$0750–\$0777	1872–1911
23	\$07D0–\$07F7	2000–2039

**Table C-2. Secondary-Page Memory Addresses**

Line	Hex Range	Dec Range
0	\$0800–\$0827	2048–2087
1	\$0880–\$08A7	2176–2215
2	\$0900–\$0927	2304–2343
3	\$0980–\$09A7	2432–2471
4	\$0A00–\$0A27	2560–2599
5	\$0A80–\$0AA7	2688–2727
6	\$0B00–\$0B27	2816–2855
7	\$0B80–\$0BA7	2944–2983
8	\$0828–\$084F	2088–2127
9	\$08A8–\$08CF	2216–2255
10	\$0928–\$094F	2344–2383
11	\$09A8–\$09CF	2472–2511
12	\$0A28–\$0A4F	2600–2639
13	\$0AA8–\$0ACF	2728–2767
14	\$0B28–\$0B4F	2856–2895
15	\$0BA8–\$0BCF	2984–3023
16	\$0850–\$0877	2128–2167
17	\$08D0–\$08F7	2256–2295
18	\$0950–\$0977	2384–2423
19	\$09D0–\$09F7	2512–2551
20	\$0A50–\$0A77	2640–2679
21	\$0AD0–\$0AF7	2768–2807
22	\$0B50–\$0B77	2896–2935
23	\$0BD0–\$0BF7	3024–3063

# Appendix

## Codes Generated by Keystrokes

These tables indicate the hexadecimal and decimal codes that are generated by a keystroke from the keyboard. **D**

**Table D-1. Hexadecimal and Decimal Key Codes (Ordinary Keys)**

Keystroke	Hex	Dec
@	\$C0	192
A	\$C1	193
B	\$C2	194
C	\$C3	195
D	\$C4	196
E	\$C4	197
F	\$C5	198
G	\$C7	199
H	\$C8	200
I	\$C9	201
J	\$CA	202
K	\$CB	203
L	\$CC	204
M	\$CD	205
N	\$CE	206
O	\$CF	207
P	\$D0	208



**Table D-1—cont. Hexadecimal and Decimal Key Codes (Ordinary Keys)**

Keystroke	Hex	Dec
Q	\$D1	209
R	\$D2	210
S	\$D3	211
T	\$D4	212
U	\$D5	213
V	\$D6	214
W	\$D7	215
X	\$D8	216
Y	\$D9	217
Z	\$DA	218
space	\$A0	160
!	\$A1	161
''	\$A2	162
#	\$A3	163
\$	\$A4	164
%	\$A5	165
&	\$A6	166
'	\$A7	167
(	\$A8	168
)	\$A9	169
*	\$AA	170
+	\$AB	171
,	\$AC	172
-	\$AD	173
.	\$AE	174
/	\$AF	175
0	\$B0	176
1	\$B1	177
2	\$B2	178
3	\$B3	179
4	\$B4	180
5	\$B5	181
6	\$B6	182
7	\$B7	183
8	\$B8	184
9	\$B9	185

**Table D-1—cont. Hexadecimal and Decimal Key Codes (Ordinary Keys)**

Keystroke	Hex	Dec
:	\$BA	186
;	\$BB	187
<	\$BC	188
=	\$BD	189
>	\$BE	190
?	\$BF	191
←	\$88	136 (Same as CTRL-H)
RET	\$8D	141 (Same as CTRL-M)
→	\$95	149 (Same as CTRL-U)
ESC	\$9B	155
^		
	\$DE	222

**Table D-2. Hexadecimal and Decimal Key Codes (Control Keys)**

<b>Keystroke</b>	<b>Hex</b>	<b>Dec</b>	
CTRL-@	\$80	128	
CTRL-A	\$81	129	
CTRL-B	\$82	130	
CTRL-C	\$83	131	
CTRL-D	\$84	132	
CTRL-E	\$85	133	
CTRL-F	\$86	134	
CTRL-G	\$87	135	
CTRL-H	\$88	136	(Same as ←)
CTRL-I	\$89	137	
CTRL-J	\$8A	138	
CTRL-K	\$8B	139	
CTRL-L	\$8C	140	
CTRL-M	\$8D	141	(Same as RETURN)
CTRL-N	\$8E	142	
CTRL-O	\$8F	143	
CTRL-P	\$90	144	
CTRL-Q	\$91	145	
CTRL-R	\$92	146	
CTRL-S	\$93	147	
CTRL-T	\$94	148	
CTRL-U	\$95	149	(Same as →)
CTRL-V	\$96	150	
CTRL-W	\$97	151	
CTRL-X	\$98	152	
CTRL-Y	\$99	153	
CTRL-Z	\$9A	154	

# Appendix

## Low-Resolution Graphics Colors

The normal low-resolution plotting operations plot a color to one-half of a character space on the screen. The colors and their codes for such operations are shown here in Table E-1.

**E**

Tables E-2 and E-3 show the two-color combinations that result from POKEing or loading codes directly to the primary or secondary page of low-resolution video memory.

**Table E-1. Low-Resolution Color Codes**

Color	Hex	Dec
BLACK	\$00	0
MAGENTA	\$01	1
DARK BLUE	\$02	2
PURPLE	\$03	3
DARK GREEN	\$04	4
GREY 1	\$05	5
MEDIUM BLUE	\$06	6
LIGHT BLUE	\$07	7
BROWN	\$08	8
ORANGE	\$09	9
GREY 2	\$0A	10
PINK	\$0B	11
LIGHT GREEN	\$0C	12
YELLOW	\$0D	13
AQUA	\$0E	14
WHITE	\$0F	15

**Table E-2. Low-Resolution Upper/Low Color Codes**

Upper/Lower	Hex	Dec
BLACK/BLACK	\$00	0
BLACK/MAGENTA	\$10	16
BLACK/DARK BLUE	\$20	32
BLACK/PURPLE	\$30	48
BLACK/DARK GREEN	\$40	64
BLACK/GREY 1	\$50	80
BLACK/MEDIUM BLUE	\$60	96
BLACK/LIGHT BLUE	\$70	112
BLACK/BROWN	\$80	128
BLACK/ORANGE	\$90	144
BLACK/GREY 2	\$A0	160
BLACK/PINK	\$B0	176
BLACK/LIGHT GREEN	\$C0	192
BLACK/YELLOW	\$D0	208
BLACK/AQUA	\$E0	224
BLACK/WHITE	\$F0	240
MAGENTA/BLACK	\$01	1
MAGENTA/MAGENTA	\$11	17
MAGENTA/DARK BLUE	\$21	33
MAGENTA/PURPLE	\$31	49
MAGENTA/DARK GREEN	\$41	65
MAGENTA/GREY 1	\$51	81
MAGENTA/MEDIUM BLUE	\$61	97
MAGENTA/LIGHT BLUE	\$71	113
MAGENTA/BROWN	\$81	129
MAGENTA/ORANGE	\$91	145
MAGENTA/GREY 2	\$A1	161
MAGENTA/PINK	\$B1	177
MAGENTA/LIGHT GREEN	\$C1	193
MAGENTA/YELLOW	\$D1	209
MAGENTA/AQUA	\$E1	225
MAGENTA/WHITE	\$F1	241
DARK BLUE/BLACK	\$02	2
DARK BLUE/MAGENTA	\$12	18
DARK BLUE/DARK BLUE	\$22	34
DARK BLUE/PURPLE	\$32	50
DARK BLUE/DARK GREEN	\$42	66
DARK BLUE/GREY 1	\$52	82
DARK BLUE/MEDIUM BLUE	\$62	98

**Table E-2—cont. Low-Resolution Upper/Low Color Codes**

<b>Upper/Lower</b>	<b>Hex</b>	<b>Dec</b>
DARK BLUE/LIGHT BLUE	\$72	114
DARK BLUE/BROWN	\$82	130
DARK BLUE/ORANGE	\$92	146
DARK BLUE/GREY 2	\$A2	162
DARK BLUE/PINK	\$B2	178
DARK BLUE/LIGHT GREEN	\$C2	194
DARK BLUE/YELLOW	\$D2	210
DARK BLUE/AQUA	\$E2	226
DARK BLUE/WHITE	\$F2	242
PURPLE/BLACK	\$03	3
PURPLE/MAGENTA	\$13	19
PURPLE/DARK BLUE	\$23	35
PURPLE/PURPLE	\$33	51
PURPLE/DARK GREEN	\$43	67
PURPLE/GREY 1	\$53	83
PURPLE/MEDIUM BLUE	\$63	99
PURPLE/LIGHT BLUE	\$73	115
PURPLE/BROWN	\$83	131
PURPLE/ORANGE	\$93	147
PURPLE/GREY 2	\$A3	163
PURPLE/PINK	\$B3	179
PURPLE/LIGHT GREEN	\$C3	195
PURPLE/YELLOW	\$D3	211
PURPLE/AQUA	\$E3	227
PURPLE/WHITE	\$F3	243
DARK GREEN BLACK	\$04	4
DARK GREEN/MAGENTA	\$14	20
DARK GREEN/DARK BLUE	\$24	36
DARK GREEN/PURPLE	\$34	52
DARK GREEN/DARK GREEN	\$44	68
DARK GREEN/GREY 1	\$54	84
DARK GREEN/MEDIUM BLUE	\$64	100
DARK GREEN/LIGHT BLUE	\$74	116
DARK GREEN/BROWN	\$84	132
DARK GREEN/ORANGE	\$94	148
DARK GREEN/GREY 2	\$A4	164
DARK GREEN/PINK	\$B4	180
DARK GREEN/LIGHT GREEN	\$C4	196
DARK GREEN/YELLOW	\$D4	212

**Table E-2—cont. Low-Resolution Upper/Low Color Codes**

<b>Upper/Lower</b>	<b>Hex</b>	<b>Dec</b>
DARK GREEN/AQUA	\$E4	228
DARK GREEN/WHITE	\$F4	244
GREY 1/BLACK	\$05	5
GREY 1/MAGENTA	\$15	21
GREY 1/DARK BLUE	\$25	37
GREY 1/PURPLE	\$35	53
GREY 1/DARK GREEN	\$45	69
GREY 1/GREY 1	\$55	85
GREY 1/MEDIUM BLUE	\$65	101
GREY 1/LIGHT BLUE	\$75	117
GREY 1/BROWN	\$85	133
GREY 1/ORANGE	\$95	149
GREY 1/GREY 2	\$A5	165
GREY 1/PINK	\$B5	181
GREY 1/LIGHT GREEN	\$C5	197
GREY 1/YELLOW	\$D5	213
GREY 1/AQUA	\$E5	229
GREY 1/WHITE	\$F5	245
MEDIUM BLUE/BLACK	\$06	6
MEDIUM BLUE/MAGENTA	\$16	22
MEDIUM BLUE/DARK BLUE	\$26	38
MEDIUM BLUE/PURPLE	\$36	54
MEDIUM BLUE/DARK GREEN	\$46	70
MEDIUM BLUE/GREY 1	\$56	86
MEDIUM BLUE/MEDIUM BLUE	\$66	102
MEDIUM BLUE/LIGHT BLUE	\$76	118
MEDIUM BLUE/BROWN	\$86	134
MEDIUM BLUE/ORANGE	\$96	150
MEDIUM BLUE/GREY 2	\$A6	166
MEDIUM BLUE/PINK	\$B6	182
MEDIUM BLUE/LIGHT GREEN	\$C6	198
MEDIUM BLUE/YELLOW	\$D6	214
MEDIUM BLUE/AQUA	\$E6	230
MEDIUM BLUE/WHITE	\$F6	246
LIGHT BLUE/BLACK	\$07	7
LIGHT BLUE/MAGENTA	\$17	23
LIGHT BLUE/DARK BLUE	\$27	39
LIGHT BLUE/PURPLE	\$37	55

**Table E-2—cont. Low-Resolution Upper/Low Color Codes**

<b>Upper/Lower</b>	<b>Hex</b>	<b>Dec</b>
LIGHT BLUE/DARK GREEN	\$47	71
LIGHT BLUE/GREY 1	\$57	87
LIGHT BLUE/MEDIUM BLUE	\$67	103
LIGHT BLUE/LIGHT BLUE	\$77	119
LIGHT BLUE/BROWN	\$87	135
LIGHT BLUE/ORANGE	\$97	151
LIGHT BLUE/GREY 2	\$A7	167
LIGHT BLUE/PINK	\$B7	183
LIGHT BLUE/LIGHT GREEN	\$C7	199
LIGHT BLUE/YELLOW	\$D7	215
LIGHT BLUE/AQUA	\$E7	231
LIGHT BLUE/WHITE	\$F7	247
BROWN/BLACK	\$08	8
BROWN/MAGENTA	\$18	24
BROWN/DARK BLUE	\$28	40
BROWN/PURPLE	\$38	56
BROWN/DARK GREEN	\$48	72
BROWN/GREY 1	\$58	88
BROWN/MEDIUM BLUE	\$68	104
BROWN/LIGHT BLUE	\$78	120
BROWN/BROWN	\$88	136
BROWN/ORANGE	\$98	152
BROWN/GREY 2	\$A8	168
BROWN/PINK	\$B8	184
BROWN/LIGHT GREEN	\$C8	200
BROWN/YELLOW	\$D8	216
BROWN/AQUA	\$E8	232
BROWN/WHITE	\$F8	248
ORANGE/BLACK	\$09	9
ORANGE/MAGENTA	\$19	25
ORANGE/DARK BLUE	\$29	41
ORANGE/PURPLE	\$39	57
ORANGE/DARK GREEN	\$49	73
ORANGE/GREY 1	\$59	89
ORANGE/MEDIUM BLUE	\$69	105
ORANGE/LIGHT BLUE	\$79	121
ORANGE/BROWN	\$89	137
ORANGE/ORANGE	\$99	153
ORANGE/GREY 2	\$A9	169



**Table E-2—cont. Low-Resolution Upper/Low Color Codes**

<b>Upper/Lower</b>	<b>Hex</b>	<b>Dec</b>
ORANGE/PINK	\$B9	185
ORANGE/LIGHT GREEN	\$C9	201
ORANGE/YELLOW	\$D9	217
ORANGE/AQUA	\$E9	233
ORANGE/WHITE	\$F9	249
GREY 2/BLACK	\$0A	10
GREY 2/MAGENTA	\$1A	26
GREY 2/DARK BLUE	\$2A	42
GREY 2/PURPLE	\$3A	58
GREY 2/DARK GREEN	\$4A	74
GREY 2/GREY 1	\$5A	90
GREY 2/MEDIUM BLUE	\$6A	106
GREY 2/LIGHT BLUE	\$7A	122
GREY 2/BROWN	\$8A	138
GREY 2/ORANGE	\$9A	154
GREY 2/GREY 2	\$AA	170
GREY 2/PINK	\$BA	186
GREY 2/LIGHT GREEN	\$CA	202
GREY 2/YELLOW	\$DA	218
GREY 2/AQUA	\$EA	234
GREY 2/WHITE	\$FA	250
PINK/BLACK	\$0B	11
PINK/MAGENTA	\$1B	27
PINK/DARK BLUE	\$2B	43
PINK/PURPLE	\$3B	59
PINK/DARK GREEN	\$4B	75
PINK/GREY 1	\$5B	91
PINK/MEDIUM BLUE	\$6B	107
PINK/LIGHT BLUE	\$7B	123
PINK/BROWN	\$8B	139
PINK/ORANGE	\$9B	155
PINK/GREY 2	\$AB	171
PINK/PINK	\$BB	187
PINK/LIGHT GREEN	\$CB	203
PINK/YELLOW	\$DB	219
PINK/AQUA	\$EB	235
PINK/WHITE	\$FB	251
LIGHT GREEN/BLACK	\$0C	12

**Table E-2—cont. Low-Resolution Upper/Low Color Codes**

Upper/Lower	Hex	Dec
LIGHT GREEN/MAGENTA	\$1C	28
LIGHT GREEN/DARK BLUE	\$2C	44
LIGHT GREEN/PURPLE	\$3C	60
LIGHT GREEN/DARK GREEN	\$4C	76
LIGHT GREEN/GREY 1	\$5C	92
LIGHT GREEN/MEDIUM BLUE	\$6C	108
LIGHT GREEN/LIGHT BLUE	\$7C	124
LIGHT GREEN/BROWN	\$8C	140
LIGHT GREEN/ORANGE	\$9C	156
LIGHT GREEN/GREY 2	\$AC	172
LIGHT GREEN/PINK	\$BC	188
LIGHT GREEN/LIGHT GREEN	\$CC	204
LIGHT GREEN/YELLOW	\$DC	220
LIGHT GREEN/AQUA	\$EC	236
LIGHT GREEN/WHITE	\$FC	252
YELLOW/BLACK	\$0D	13
YELLOW/MAGENTA	\$1D	29
YELLOW/DARK BLUE	\$2D	45
YELLOW/PURPLE	\$3D	61
YELLOW/DARK GREEN	\$4D	77
YELLOW/GREY 1	\$5D	93
YELLOW/MEDIUM BLUE	\$6D	109
YELLOW/LIGHT BLUE	\$7D	125
YELLOW/BROWN	\$8D	141
YELLOW/ORANGE	\$9D	157
YELLOW/GREY 2	\$AD	173
YELLOW/PINK	\$BD	189
YELLOW/LIGHT GREEN	\$CD	205
YELLOW/YELLOW	\$DD	221
YELLOW/AQUA	\$ED	237
YELLOW/WHITE	\$FD	253
AQUA/BLACK	\$0E	14
AQUA/MAGENTA	\$1E	30
AQUA/DARK BLUE	\$2E	46
AQUA/PURPLE	\$3E	62
AQUA/DARK GREEN	\$4E	78
AQUA/GREY 1	\$5E	94
AQUA/MEDIUM BLUE	\$6E	110
AQUA/LIGHT BLUE	\$7E	126

**Table E-2—cont. Low-Resolution Upper/Low Color Codes**

Upper/Lower	Hex	Dec
AQUA/BROWN	\$8E	142
AQUA/ORANGE	\$9E	158
AQUA/GREY 2	\$AE	174
AQUA/PINK	\$BE	190
AQUA/LIGHT GREEN	\$CE	206
AQUA/YELLOW	\$DE	222
AQUA/AQUA	\$EE	238
AQUA/WHITE	\$FE	254
WHITE/BLACK	\$0F	15
WHITE/MAGENTA	\$1F	31
WHITE/DARK BLUE	\$2F	47
WHITE/PURPLE	\$3F	63
WHITE/DARK GREEN	\$4F	79
WHITE/GREY 1	\$5F	95
WHITE/MEDIUM BLUE	\$6F	111
WHITE/LIGHT BLUE	\$7F	127
WHITE/BROWN	\$8F	143
WHITE/ORANGE	\$9F	159
WHITE/GREY 2	\$AF	175
WHITE/PINK	\$BF	191
WHITE/LIGHT GREEN	\$CF	207
WHITE/YELLOW	\$DF	223
WHITE/AQUA	\$EF	239
WHITE/WHITE	\$FF	255

**Table E-3. Low-Resolution Lower/Upper Color Codes**

Lower/Upper	Hex	Dec
BLACK/BLACK	\$00	0
BLACK/MAGENTA	\$01	1
BLACK/DARK BLUE	\$02	2
BLACK/PURPLE	\$03	3
BLACK/DARK GREEN	\$04	4
BLACK/GREY 1	\$05	5
BLACK/MEDIUM BLUE	\$06	6
BLACK/LIGHT BLUE	\$07	7
BLACK/BROWN	\$08	8
BLACK/ORANGE	\$09	9
BLACK/GREY 2	\$0A	10
BLACK/PINK	\$0B	11
BLACK/LIGHT GREEN	\$0C	12

**Table E-3—cont. Low-Resolution Lower/Upper Color Codes**

<b>Lower/Upper</b>	<b>Hex</b>	<b>Dec</b>
BLACK/YELLOW	\$0D	13
BLACK/AQUA	\$0E	14
BLACK/WHITE	\$0F	15
MAGENTA/BLACK	\$10	16
MAGENTA/MAGENTA	\$11	17
MAGENTA/DARK BLUE	\$12	18
MAGENTA/PURPLE	\$13	19
MAGENTA/DARK GREEN	\$14	20
MAGENTA/GREY 1	\$15	21
MAGENTA/MEDIUM BLUE	\$16	22
MAGENTA/LIGHT BLUE	\$17	23
MAGENTA/BROWN	\$18	24
MAGENTA/ORANGE	\$19	25
MAGENTA/GREY 2	\$1A	26
MAGENTA/PINK	\$1B	27
MAGENTA/LIGHT GREEN	\$1C	28
MAGENTA/YELLOW	\$1D	29
MAGENTA/AQUA	\$1E	30
MAGENTA/WHITE	\$1F	31
DARK BLUE/BLACK	\$20	32
DARK BLUE/MAGENTA	\$21	33
DARK BLUE/DARK BLUE	\$22	34
DARK BLUE/PURPLE	\$23	35
DARK BLUE/DARK GREEN	\$24	36
DARK BLUE/GREY 1	\$25	37
DARK BLUE/MEDIUM BLUE	\$26	38
DARK BLUE/LIGHT BLUE	\$27	39
DARK BLUE/BROWN	\$28	40
DARK BLUE/ORANGE	\$29	41
DARK BLUE/GREY 2	\$2A	42
DARK BLUE/PINK	\$2B	43
DARK BLUE/LIGHT GREEN	\$2C	44
DARK BLUE/YELLOW	\$2D	45
DARK BLUE/AQUA	\$2E	46
DARK BLUE/WHITE	\$2F	47
PURPLE/BLACK	\$30	48
PURPLE/MAGENTA	\$31	49
PURPLE/DARK BLUE	\$32	50

**Table E-3—cont. Low-Resolution Lower/Upper Color Codes**

Lower/Upper	Hex	Dec
PURPLE/PURPLE	\$33	51
PURPLE/DARK GREEN	\$34	52
PURPLE/GREY 1	\$35	53
PURPLE/MEDIUM BLUE	\$36	54
PURPLE/LIGHT BLUE	\$37	55
PURPLE/BROWN	\$38	56
PURPLE/ORANGE	\$39	57
PURPLE/GREY 2	\$3A	58
PURPLE/PINK	\$3B	59
PURPLE/LIGHT GREEN	\$3C	60
PURPLE/YELLOW	\$3D	61
PURPLE/AQUA	\$3E	62
PURPLE/WHITE	\$3F	63
DARK GREEN/BLACK	\$40	64
DARK GREEN/MAGENTA	\$41	65
DARK GREEN/DARK BLUE	\$42	66
DARK GREEN/PURPLE	\$43	67
DARK GREEN/DARK GREEN	\$44	68
DARK GREEN/GREY 1	\$45	69
DARK GREEN/MEDIUM BLUE	\$46	70
DARK GREEN/LIGHT BLUE	\$47	71
DARK GREEN/BROWN	\$48	72
DARK GREEN/ORANGE	\$49	73
DARK GREEN/GREY 2	\$4A	74
DARK GREEN/PINK	\$4B	75
DARK GREEN/LIGHT GREEN	\$4C	76
DARK GREEN/YELLOW	\$4D	77
DARK GREEN/AQUA	\$4E	78
DARK GREEN/WHITE	\$4F	79
GREY 1/BLACK	\$50	80
GREY 1/MAGENTA	\$51	81
GREY 1/DARK BLUE	\$52	82
GREY 1/PURPLE	\$53	83
GREY 1/DARK GREEN	\$54	84
GREY 1/GREY 1	\$55	85
GREY 1/MEDIUM BLUE	\$56	86
GREY 1/LIGHT BLUE	\$57	87
GREY 1/BROWN	\$58	88
GREY 1/ORANGE	\$59	89

**Table E-3—cont. Low-Resolution Lower/Upper Color Codes**

Lower/Upper	Hex	Dec
GREY 1/GREY 2	\$5A	90
GREY 1/PINK	\$5B	91
GREY 1/LIGHT GREEN	\$5C	92
GREY 1/YELLOW	\$5D	93
GREY 1/AQUA	\$5E	94
GREY 1/WHITE	\$5F	95
MEDIUM BLUE/BLACK	\$60	96
MEDIUM BLUE/MAGENTA	\$61	97
MEDIUM BLUE/DARK BLUE	\$62	98
MEDIUM BLUE/PURPLE	\$63	99
MEDIUM BLUE/DARK GREEN	\$64	100
MEDIUM BLUE/GREY 1	\$65	101
MEDIUM BLUE/MEDIUM BLUE	\$66	102
MEDIUM BLUE/LIGHT BLUE	\$67	103
MEDIUM BLUE/BROWN	\$68	104
MEDIUM BLUE/ORANGE	\$69	105
MEDIUM BLUE/GREY 2	\$6A	106
MEDIUM BLUE/PINK	\$6B	107
MEDIUM BLUE/LIGHT GREEN	\$6C	108
MEDIUM BLUE/YELLOW	\$6D	109
MEDIUM BLUE/AQUA	\$6E	110
MEDIUM BLUE/WHITE	\$6F	111
LIGHT BLUE/BLACK	\$70	112
LIGHT BLUE/MAGENTA	\$71	113
LIGHT BLUE/DARK BLUE	\$72	114
LIGHT BLUE/PURPLE	\$73	115
LIGHT BLUE/DARK GREEN	\$74	116
LIGHT BLUE/GREY 1	\$75	117
LIGHT BLUE/MEDIUM BLUE	\$76	118
LIGHT BLUE/LIGHT BLUE	\$77	119
LIGHT BLUE/BROWN	\$78	120
LIGHT BLUE/ORANGE	\$79	121
LIGHT BLUE/GREY 2	\$7A	122
LIGHT BLUE/PINK	\$7B	123
LIGHT BLUE/LIGHT GREEN	\$7C	124
LIGHT BLUE/YELLOW	\$7D	125
LIGHT BLUE/AQUA	\$7E	126
LIGHT BLUE/WHITE	\$7F	127

**Table E-3—cont. Low-Resolution Lower/Upper Color Codes**

Lower/Upper	Hex	Dec
BROWN/BLACK	\$80	128
BROWN/MAGENTA	\$81	129
BROWN/DARK BLUE	\$82	130
BROWN/PURPLE	\$83	131
BROWN/DARK GREEN	\$84	132
BROWN/GREY 1	\$85	133
BROWN/MEDIUM BLUE	\$86	134
BROWN/LIGHT BLUE	\$87	135
BROWN/BROWN	\$88	136
BROWN/ORANGE	\$89	137
BROWN/GREY 2	\$8A	138
BROWN/PINK	\$8B	139
BROWN/LIGHT GREEN	\$8C	140
BROWN/YELLOW	\$8D	141
BROWN/AQUA	\$8E	142
BROWN/WHITE	\$8F	143
ORANGE/BLACK	\$90	144
ORANGE/MAGENTA	\$91	145
ORANGE/DARK BLUE	\$92	146
ORANGE/PURPLE	\$93	147
ORANGE/DARK GREEN	\$94	148
ORANGE/GREY 1	\$95	149
ORANGE/MEDIUM BLUE	\$96	150
ORANGE/LIGHT BLUE	\$97	151
ORANGE/BROWN	\$98	152
ORANGE/ORANGE	\$99	153
ORANGE/GREY 2	\$9A	154
ORANGE/PINK	\$9B	155
ORANGE/LIGHT GREEN	\$9C	156
ORANGE/YELLOW	\$9D	157
ORANGE/AQUA	\$9E	158
ORANGE/WHITE	\$9F	159
GREY 2/BLACK	\$A0	160
GREY 2/MAGENTA	\$A1	161
GREY 2/DARK BLUE	\$A2	162
GREY 2/PURPLE	\$A3	163
GREY 2/DARK GREEN	\$A4	164
GREY 2/GREY 1	\$A5	165
GREY 2/MEDIUM BLUE	\$A6	166

**Table E-3—cont. Low-Resolution Lower/Upper Color Codes**

<b>Lower/Upper</b>	<b>Hex</b>	<b>Dec</b>
GREY 2/LIGHT BLUE	\$A7	167
GREY 2/BROWN	\$A8	168
GREY 2/ORANGE	\$A9	169
GREY 2/GREY 2	\$AA	170
GREY 2/PINK	\$AB	171
GREY 2/LIGHT GREEN	\$AC	172
GREY 2/YELLOW	\$AD	173
GREY 2/AQUA	\$AE	174
GREY 2/WHITE	\$AF	175
PINK/BLACK	\$B0	176
PINK/MAGENTA	\$B1	177
PINK/DARK BLUE	\$B2	178
PINK/PURPLE	\$B3	179
PINK/DARK GREEN	\$B4	180
PINK/GREY 1	\$B5	181
PINK/MEDIUM BLUE	\$B6	182
PINK/LIGHT BLUE	\$B7	183
PINK/BROWN	\$B8	184
PINK/ORANGE	\$B9	185
PINK/GREY 2	\$BA	186
PINK/PINK	\$BB	187
PINK/LIGHT GREEN	\$BC	188
PINK/YELLOW	\$BD	189
PINK/AQUA	\$BE	190
PINK/WHITE	\$BF	191
LIGHT GREEN/BLACK	\$C0	192
LIGHT GREEN/MAGENTA	\$C1	193
LIGHT GREEN/DARK BLUE	\$C2	194
LIGHT GREEN/PURPLE	\$C3	195
LIGHT GREEN/DARK GREEN	\$C4	196
LIGHT GREEN/GREY 1	\$C5	197
LIGHT GREEN/MEDIUM BLUE	\$C6	198
LIGHT GREEN/LIGHT BLUE	\$C7	199
LIGHT GREEN/BROWN	\$C8	200
LIGHT GREEN/ORANGE	\$C9	201
LIGHT GREEN/GREY 2	\$CA	202
LIGHT GREEN/PINK	\$CB	203
LIGHT GREEN/LIGHT GREEN	\$CC	204
LIGHT GREEN/YELLOW	\$CD	205



**Table E-3—cont. Low-Resolution Lower/Upper Color Codes**

Lower/Upper	Hex	Dec
LIGHT GREEN/AQUA	\$CE	206
LIGHT GREEN/WHITE	\$CF	207
YELLOW/BLACK	\$D0	208
YELLOW/MAGENTA	\$D1	209
YELLOW/DARK BLUE	\$D2	210
YELLOW/PURPLE	\$D3	211
YELLOW/DARK GREEN	\$D4	212
YELLOW/GREY 1	\$D5	213
YELLOW/MEDIUM BLUE	\$D6	214
YELLOW/LIGHT BLUE	\$D7	215
YELLOW/BROWN	\$D8	216
YELLOW/ORANGE	\$D9	217
YELLOW/GREY 2	\$DA	218
YELLOW/PINK	\$DB	219
YELLOW/LIGHT GREEN	\$DC	220
YELLOW/YELLOW	\$DD	221
YELLOW/AQUA	\$DE	222
YELLOW/WHITE	\$DF	223
AQUA/BLACK	\$E0	224
AQUA/MAGENTA	\$E1	225
AQUA/DARK BLUE	\$E2	226
AQUA/PURPLE	\$E3	227
AQUA/DARK GREEN	\$E4	228
AQUA/GREY 1	\$E5	229
AQUA/MEDIUM BLUE	\$E6	230
AQUA/LIGHT BLUE	\$E7	231
AQUA/BROWN	\$E8	232
AQUA/ORANGE	\$E9	233
AQUA/GREY 2	\$EA	234
AQUA/PINK	\$EB	235
AQUA/LIGHT GREEN	\$EC	236
AQUA/YELLOW	\$ED	237
AQUA/AQUA	\$EE	238
AQUA/WHITE	\$EF	239
WHITE/BLACK	\$F0	240
WHITE/MAGENTA	\$F1	241
WHITE/DARK BLUE	\$F2	242
WHITE/PURPLE	\$F3	243

**Table E-3—cont. Low-Resolution Lower/Upper Color Codes**

<b>Lower/Upper</b>	<b>Hex</b>	<b>Dec</b>
WHITE/DARK GREEN	\$F4	244
WHITE/GREY 1	\$F5	245
WHITE/MEDIUM BLUE	\$F6	246
WHITE/LIGHT BLUE	\$F7	247
WHITE/BROWN	\$F8	248
WHITE/ORANGE	\$F9	249
WHITE/GREY 2	\$FA	250
WHITE/PINK	\$FB	251
WHITE/LIGHT GREEN	\$FC	252
WHITE/YELLOW	\$FD	253
WHITE/AQUA	\$FE	254
WHITE/WHITE	\$FF	255

# Appendix

## Range of High-Resolution Graphics Video Addresses

These tables show the range of hexadecimal and decimal addresses for each of the 192 high-resolution graphics lines. The first table represents the primary page, and the second table shows the ranges for the secondary high-resolution page.

**Table F-1. High-Resolution Primary-Page Addresses**

Line	Hex Range	Dec Range
0	\$2000-\$2027	8192-8231
1	\$2400-\$2427	9216-9255
2	\$2800-\$2827	10240-10279
3	\$2C00-\$2C27	11264-11303
4	\$3000-\$3027	12288-12327
5	\$3400-\$3427	13312-13351
6	\$3800-\$3827	14336-14375
7	\$3C00-\$3C27	15360-15399
8	\$2080-\$20A7	8320-8359
9	\$2480-\$24A7	9344-9383
10	\$2880-\$28A7	10368-10407
11	\$2C80-\$2CA7	11392-11431
12	\$3080-\$30A7	12416-12455
13	\$3480-\$34A7	13440-13479
14	\$3880-\$38A7	14464-14503

**Table F-1—cont. High-Resolution Primary-Page Addresses**

<b>Line</b>	<b>Hex Range</b>	<b>Dec Range</b>
15	\$3C80–\$3CA7	15488–15527
16	\$2100–\$2127	8448–8487
17	\$2500–\$2527	9472–9511
18	\$2900–\$2927	10496–10535
19	\$2D00–\$2D27	11520–11559
20	\$3100–\$3127	12544–12583
21	\$3500–\$3527	13568–13607
22	\$3900–\$3927	14592–14631
23	\$3D00–\$3D27	15616–15655
24	\$2180–\$21A7	8576–8615
25	\$2580–\$25A7	9600–9639
26	\$2980–\$29A7	10624–10663
27	\$2D80–\$2DA7	11648–11687
28	\$3180–\$31A7	12672–12711
29	\$3580–\$35A7	13696–13735
30	\$3980–\$39A7	14720–14759
31	\$3D80–\$3DA7	15744–15783
32	\$2200–\$2227	8704–8743
33	\$2600–\$2627	9728–9767
34	\$2A00–\$2A27	10752–10791
35	\$2E00–\$2E27	11776–11815
36	\$3200–\$3227	12800–12839
37	\$3600–\$3627	13824–13863
38	\$3A00–\$3A27	14848–14887
39	\$3E00–\$3E27	15872–15911
40	\$2280–\$22A7	8832–8871
41	\$2680–\$26A7	9856–9895
42	\$2A80–\$2AA7	10880–10919
43	\$2E80–\$2EA7	11904–11943
44	\$3280–\$32A7	12928–12967
45	\$3680–\$36A7	13952–13991
46	\$3A80–\$3AA7	14976–15015
47	\$3E80–\$3EA7	16000–16039
48	\$2300–\$2327	8960–8999
49	\$2700–\$2727	9984–10023
50	\$2B00–\$2B27	11008–11047

**Table F-1—cont. High-Resolution Primary-Page Addresses**

Line	Hex Range	Dec Range
51	\$2F00–\$2F27	12032–12071
52	\$3300–\$3327	13056–13095
53	\$3700–\$3727	14080–14119
54	\$3B00–\$3B27	15104–15143
55	\$3F00–\$3F27	16128–16167
56	\$2380–\$23A7	9088–9127
57	\$2780–\$27A7	10112–10151
58	\$2B80–\$2BA7	11136–11175
59	\$2F80–\$2FA7	12160–12199
60	\$3380–\$33A7	13184–13223
61	\$3780–\$37A7	14208–14247
62	\$3B80–\$3BA7	15232–15271
63	\$3F80–\$3FA7	16256–16295
64	\$2028–\$204F	8232–8271
65	\$2428–\$244F	9256–9295
66	\$2828–\$284F	10280–10319
67	\$2C28–\$2C4F	11304–11343
68	\$3028–\$304F	12328–12367
69	\$3428–\$344F	13352–13391
70	\$3828–\$384F	14376–14415
71	\$3C28–\$3C4F	15400–15439
72	\$20A8–\$20CF	8360–8399
73	\$24A8–\$24CF	9384–9423
74	\$28A8–\$28CF	10408–10447
75	\$2CA8–\$2CCF	11432–11471
76	\$30A8–\$30CF	12456–12495
77	\$34A8–\$34CF	13480–13519
78	\$38A8–\$38CF	14504–14543
79	\$3CA8–\$3CCF	15528–15567
80	\$2128–\$214F	8488–8527
81	\$2528–\$254F	9512–9551
82	\$2928–\$294F	10536–10575
83	\$2D28–\$2D4F	11560–11599
84	\$3128–\$314F	12584–12623
85	\$3528–\$354F	13608–13647
86	\$3928–\$394F	14632–14671
87	\$3D28–\$3D4F	15656–15695

**Table F-1—cont. High-Resolution Primary-Page Addresses**

<b>Line</b>	<b>Hex Range</b>	<b>Dec Range</b>
88	\$21A8–\$21CF	8616–8655
89	\$25A8–\$25CF	9640–9679
90	\$29A8–\$29CF	10664–10703
91	\$2DA8–\$2DCF	11688–11727
92	\$31A8–\$31CF	12712–12751
93	\$35A8–\$35CF	13736–13775
94	\$39A8–\$39CF	14760–14799
95	\$3DA8–\$3DCF	15784–15823
96	\$2228–\$224F	8744–8783
97	\$2628–\$264F	9768–9807
98	\$2A28–\$2A4F	10792–10831
99	\$2E28–\$2E4F	11816–11855
100	\$3228–\$324F	12840–12879
101	\$3628–\$364F	13864–13903
102	\$3A28–\$3A4F	14888–14927
103	\$3E28–\$3E4F	15912–15951
104	\$22A8–\$22CF	8872–8911
105	\$26A8–\$26CF	9896–9935
106	\$2AA8–\$2ACF	10920–10959
107	\$2EA8–\$2ECF	11944–11983
108	\$32A8–\$32CF	12968–13007
109	\$36A8–\$36CF	13992–14031
110	\$3AA8–\$3ACF	15016–15055
111	\$3EA8–\$3ECF	16040–16079
112	\$2328–\$234F	9000–9039
113	\$2728–\$274F	10024–10063
114	\$2B28–\$2B4F	11048–11087
115	\$2F28–\$2F4F	12072–12111
116	\$3328–\$334F	13096–13135
117	\$3728–\$374F	14120–14159
118	\$3B28–\$3B4F	15144–15183
119	\$3F28–\$3F4F	16168–16207
120	\$23A8–\$23CF	9128–9167
121	\$27A8–\$27CF	10152–10191
122	\$2BA8–\$2BCF	11176–11215
123	\$2FA8–\$2FCF	12200–12239
124	\$33A8–\$33CF	13224–13263

**Table F-1—cont. High-Resolution Primary-Page Addresses**

Line	Hex Range	Dec Range
125	\$37A8–\$37CF	14248–14287
126	\$3BA8–\$3BCF	15272–15311
127	\$3FA8–\$3FCF	16296–16335
128	\$2050–\$2077	8272–8311
129	\$2450–\$2477	9296–9335
130	\$2850–\$2877	10320–10359
131	\$2C50–\$2C77	11344–11383
132	\$3050–\$3077	12368–12407
133	\$3450–\$3477	13392–13431
134	\$3850–\$3877	14416–14455
135	\$3C50–\$3C77	15440–15479
136	\$20D0–\$20F7	8400–8439
137	\$24D0–\$24F7	9424–9463
138	\$28D0–\$28F7	10448–10487
139	\$2CD0–\$2CF7	11472–11511
140	\$30D0–\$30F7	12496–12535
141	\$34D0–\$34F7	13520–13559
142	\$38D0–\$38F7	14544–14583
143	\$3CD0–\$3CF7	15568–15607
144	\$2150–\$2177	8528–8567
145	\$2550–\$2577	9552–9591
146	\$2950–\$2977	10576–10615
147	\$2D50–\$2D77	11600–11639
148	\$3150–\$3177	12624–12663
149	\$3550–\$3577	13648–13687
150	\$3950–\$3977	14672–14711
151	\$3D50–\$3D77	15696–15735
152	\$21D0–\$21F7	8656–8695
153	\$25D0–\$25F7	9680–9719
154	\$29D0–\$29F7	10704–10743
155	\$2DD0–\$2DF7	11728–11767
156	\$31D0–\$31F7	12752–12791
157	\$35D0–\$35F7	13776–13815
158	\$39D0–\$39F7	14800–14839
159	\$3DD0–\$3DF7	15824–15863
160	\$2250–\$2277	8784–8823
161	\$2650–\$2677	9808–9847

**Table F-1—cont. High-Resolution Primary-Page Addresses**

Line	Hex Range	Dec Range
162	\$2A50–\$2A77	10832–10871
163	\$2E50–\$2E77	11856–11895
164	\$3250–\$3277	12880–12919
165	\$3650–\$3677	13904–13943
166	\$3A50–\$3A77	14928–14967
167	\$3E50–\$3E77	15952–15991
168	\$22D0–\$22F7	8912–8951
169	\$26D0–\$26F7	9936–9975
170	\$2AD0–\$2AF7	10960–10999
171	\$2ED0–\$2EF7	11984–12023
172	\$32D0–\$32F7	13008–13047
173	\$36D0–\$36F7	14032–14071
174	\$3AD0–\$3AF7	15056–15095
175	\$3ED0–\$3EF7	16080–16119
176	\$2350–\$2377	9040–9079
177	\$2750–\$2777	10064–10103
178	\$2B50–\$2B77	11088–11127
179	\$2F50–\$2F77	12112–12151
180	\$3350–\$3377	13136–13175
181	\$3750–\$3777	14160–14199
182	\$3B50–\$3B77	15184–15223
183	\$3F50–\$3F77	16208–16247
184	\$23D0–\$23F7	9168–9207
185	\$27D0–\$27F7	10192–10231
186	\$2BD0–\$2BF7	11216–11255
187	\$2FD0–\$2FF7	12240–12279
188	\$33D0–\$33F7	13264–13303
189	\$37D0–\$37F7	14288–14327
190	\$3BD0–\$3BF7	15312–15351
191	\$3FD0–\$3FF7	16336–16375

**Table F-2. High-Resolution Secondary-Page Addresses**

Line	Hex Range	Dec Range
0	\$4000–\$4027	16384–16423
1	\$4400–\$4427	17408–17447
2	\$4800–\$4827	18432–18471
3	\$4C00–\$4C27	19456–19495



**Table F-2—cont. High-Resolution Secondary-Page Addresses**

Line	Hex Range	Dec Range
4	\$5000–\$5027	20480–20519
5	\$5400–\$5427	21504–21543
6	\$5800–\$5827	22528–22567
7	\$5C00–\$5C27	23552–23591
8	\$4080–\$40A7	16512–16551
9	\$4480–\$44A7	17536–17575
10	\$4880–\$48A7	18560–18599
11	\$4C80–\$4CA7	19584–19623
12	\$5080–\$50A7	20608–20647
13	\$5480–\$54A7	21632–21671
14	\$5880–\$58A7	22656–22695
15	\$5C80–\$5CA7	23680–23719
16	\$4100–\$4127	16640–16679
17	\$4500–\$4527	17664–17703
18	\$4900–\$4927	18688–18727
19	\$4D00–\$4D27	19712–19751
20	\$5100–\$5127	20736–20775
21	\$5500–\$5527	21760–21799
22	\$5900–\$5927	22784–22823
23	\$5D00–\$5D27	23808–23847
24	\$4180–\$41A7	16768–16807
25	\$4580–\$45A7	17792–17831
26	\$4980–\$49A7	18816–18855
27	\$4D80–\$4DA7	19840–19879
28	\$5180–\$51A7	20864–20903
29	\$5580–\$55A7	21888–21927
30	\$5980–\$59A7	22912–22951
31	\$5D80–\$5DA7	23936–23975
32	\$4200–\$4227	16896–16935
33	\$4600–\$4627	17920–17959
34	\$4A00–\$4A27	18944–18983
35	\$4E00–\$4E27	19968–20007
36	\$5200–\$5227	20992–21031
37	\$5600–\$5627	22016–22055
38	\$5A00–\$5A27	23040–23079
39	\$5E00–\$5E27	24064–24103

**Table F-2—cont. High-Resolution Secondary-Page Addresses**

Line	Hex Range	Dec Range
40	\$4280–\$42A7	17024–17063
41	\$4680–\$46A7	18048–18087
42	\$4A80–\$4AA7	19072–19111
43	\$4E80–\$4EA7	20096–20135
44	\$5280–\$52A7	21120–21159
45	\$5680–\$56A7	22144–22183
46	\$5A80–\$5AA7	23168–23207
47	\$5E80–\$5EA7	24192–24231
48	\$4300–\$4327	17152–17191
49	\$4700–\$4727	18176–18215
50	\$4B00–\$4B27	19200–19239
51	\$4F00–\$4F27	20224–20263
52	\$5300–\$5327	21248–21287
53	\$5700–\$5727	22272–22311
54	\$5B00–\$5B27	23296–23335
55	\$5F00–\$5F27	24320–24359
56	\$4380–\$43A7	17280–17319
57	\$4780–\$47A7	18304–18343
58	\$4B80–\$4BA7	19328–19367
59	\$4F80–\$4FA7	20352–20391
60	\$5380–\$53A7	21376–21415
61	\$5780–\$57A7	22400–22439
62	\$5B80–\$5BA7	23424–23463
63	\$5F80–\$5FA7	24448–24487
64	\$4028–\$404F	16424–16463
65	\$4428–\$444F	17448–17487
66	\$4828–\$484F	18472–18511
67	\$4C28–\$4C4F	19496–19535
68	\$5028–\$504F	20520–20559
69	\$5428–\$544F	21544–21583
70	\$5828–\$584F	22568–22607
71	\$5C28–\$5C4F	23592–23631
72	\$40A8–\$40CF	16552–16591
73	\$44A8–\$44CF	17576–17615
74	\$48A8–\$48CF	18600–18639
75	\$4CA8–\$4CCF	19624–19663
76	\$50A8–\$50CF	20648–20687

**Table F-2—cont. High-Resolution Secondary-Page Addresses**

<b>Line</b>	<b>Hex Range</b>	<b>Dec Range</b>
77	\$54A8–\$54CF	21672–21711
78	\$58A8–\$58CF	22696–22735
79	\$5CA8–\$5CCF	23720–23759
80	\$4128–\$414F	16680–16719
81	\$4528–\$454F	17704–17743
82	\$4928–\$494F	18728–18767
83	\$4D28–\$4D4F	19752–19791
84	\$5128–\$514F	20776–20815
85	\$5528–\$554F	21800–21839
86	\$5928–\$594F	22824–22863
87	\$5D28–\$5D4F	23848–23887
88	\$41A8–\$41CF	16808–16847
89	\$45A8–\$45CF	17832–17871
90	\$49A8–\$49CF	18856–18895
91	\$4DA8–\$4DCF	19880–19919
92	\$51A8–\$51CF	20904–20943
93	\$55A8–\$55CF	21928–21967
94	\$59A8–\$59CF	22952–22991
95	\$5DA8–\$5DCF	23976–24015
96	\$4228–\$424F	16936–16975
97	\$4628–\$464F	17960–17999
98	\$4A28–\$4A4F	18984–19023
99	\$4E28–\$4E4F	20008–20047
100	\$5228–\$524F	21032–21071
101	\$5628–\$564F	22056–22095
102	\$5A28–\$5A4F	23080–23119
103	\$5E28–\$5E4F	24104–24143
104	\$42A8–\$42CF	17064–17103
105	\$46A8–\$46CF	18088–18127
106	\$4AA8–\$4ACF	19112–19151
107	\$4EA8–\$4ECF	20136–20175
108	\$52A8–\$52CF	21160–21199
109	\$56A8–\$56CF	22184–22223
110	\$5AA8–\$5ACF	23208–23247
111	\$5EA8–\$5ECF	24232–24271
112	\$4328–\$434F	17192–17231
113	\$4728–\$474F	18216–18255

**Table F-2—cont. High-Resolution Secondary-Page Addresses**

<b>Line</b>	<b>Hex Range</b>	<b>Dec Range</b>
114	\$4B28–\$4B4F	19240–19279
115	\$4F28–\$4F4F	20264–20303
116	\$5328–\$534F	21288–21327
117	\$5728–\$574F	22312–22351
118	\$5B28–\$5B4F	23336–23375
119	\$5F28–\$5F4F	24360–24399
120	\$43A8–\$43CF	17320–17359
121	\$47A8–\$47CF	18344–18383
122	\$4BA8–\$4BCF	19368–19407
123	\$4FA8–\$4FCF	20392–20431
124	\$53A8–\$53CF	21416–21455
125	\$57A8–\$57CF	22440–22479
126	\$5BA8–\$5BCF	23464–23503
127	\$5FA8–\$5FCF	24488–24527
128	\$4050–\$4077	16464–16503
129	\$4450–\$4477	17488–17527
130	\$4850–\$4877	18512–18551
131	\$4C50–\$4C77	19536–19575
132	\$5050–\$5077	20560–20599
133	\$5450–\$5477	21584–21623
134	\$5850–\$5877	22608–22647
135	\$5C50–\$5C77	23632–23671
136	\$40D0–\$40F7	16592–16631
137	\$44D0–\$44F7	17616–17655
138	\$48D0–\$48F7	18640–18679
139	\$4CD0–\$4CF7	19664–19703
140	\$50D0–\$50F7	20688–20727
141	\$54D0–\$54F7	21712–21751
142	\$58D0–\$58F7	22736–22775
143	\$5CD0–\$5CF7	23760–23799
144	\$4150–\$4177	16720–16759
145	\$4550–\$4577	17744–17783
146	\$4950–\$4977	18768–18807
147	\$4D50–\$4D77	19792–19831
148	\$5150–\$5177	20816–20855
149	\$5550–\$5577	21840–21879

**Table F-2—cont. High-Resolution Secondary-Page Addresses**

<b>Line</b>	<b>Hex Range</b>	<b>Dec Range</b>
150	\$5950–\$5977	22864–22903
151	\$5D50–\$5D77	23888–23927
152	\$41D0–\$41F7	16848–16887
153	\$45D0–\$45F7	17872–17911
154	\$49D0–\$49F7	18896–18935
155	\$4DD0–\$4DF7	19920–19959
156	\$51D0–\$51F7	20944–20983
157	\$55D0–\$55F7	21968–22007
158	\$59D0–\$59F7	22992–23031
159	\$5DD0–\$5DF7	24016–24055
160	\$4250–\$4277	16976–17015
161	\$4650–\$4677	18000–18039
162	\$4A50–\$4A77	19024–19063
163	\$4E50–\$4E77	20048–20087
164	\$5250–\$5277	21072–21111
165	\$5650–\$5677	22096–22135
166	\$5A50–\$5A77	23120–23159
167	\$5E50–\$5E77	24144–24183
168	\$42D0–\$42F7	17104–17143
169	\$46D0–\$46F7	18128–18167
170	\$4AD0–\$4AF7	19152–19191
171	\$4ED0–\$4EF7	20176–20215
172	\$52D0–\$52F7	21200–21239
173	\$56D0–\$56F7	22224–22263
174	\$5AD0–\$5AF7	23248–23287
175	\$5ED0–\$5EF7	24272–24311
176	\$4350–\$4377	17232–17271
177	\$4750–\$4777	18256–18295
178	\$4B50–\$4B77	19280–19319
179	\$4F50–\$4F77	20304–20343
180	\$5350–\$5377	21328–21367
181	\$5750–\$5777	22352–22391
182	\$5B50–\$5B77	23376–23415
183	\$5F50–\$5F77	24400–24439
184	\$43D0–\$43F7	17360–17399
185	\$47D0–\$47F7	18384–18423

**Table F-2—cont. High-Resolution Secondary-Page Addresses**

<b>Line</b>	<b>Hex Range</b>	<b>Dec Range</b>
186	\$4BD0–\$4BF7	19408–19447
187	\$4FD0–\$4FF7	20432–20471
188	\$53D0–\$53F7	21456–21495
189	\$57D0–\$57F7	22480–22519
190	\$5BD0–\$5BF7	23504–23543
191	\$5FD0–\$5FF7	24528–24567

# Appendix G

## 6502 Instruction Set

**Table G-1. 6502 Instruction Set**

Mnemonic	Machine Language	Comments
ADS <i>#data</i>	69 <i>byte</i>	Add with carry
ADA <i>addr<sub>0</sub></i>	65 <i>byte</i>	
ADC <i>addr</i>	6D <i>byte byte</i>	
ADC <i>addr<sub>0</sub>,X</i>	75 <i>byte</i>	
ADC <i>addr,X</i>	7D <i>byte byte</i>	
ADC <i>addr,Y</i>	79 <i>byte byte</i>	
ADC ( <i>data</i> ,X)	61 <i>byte</i>	
ADC ( <i>data</i> ),Y	71 <i>byte</i>	
AND <i>#data</i>	29 <i>byte</i>	Logical AND
AND <i>addr<sub>0</sub></i>	25 <i>byte</i>	
AND <i>addr</i>	2D <i>byte byte</i>	
AND <i>addr<sub>0</sub>,X</i>	35 <i>byte</i>	
AND <i>addr,X</i>	3D <i>byte byte</i>	
AND <i>addr,Y</i>	39 <i>byte byte</i>	
AND ( <i>data</i> ,X)	21 <i>byte</i>	
AND ( <i>data</i> ),Y	31 <i>byte</i>	
ASL A	0A	Shift left
ASL <i>addr<sub>0</sub></i>	06 <i>byte</i>	
ASL <i>addr</i>	0E <i>byte byte</i>	
ASL <i>addr<sub>0</sub>,X</i>	16 <i>byte</i>	
ASL <i>addr,X</i>	1E <i>byte byte</i>	

Table G-1—cont. 6502 Instruction Set

Mnemonic	Machine Language	Comments
BCC <i>disp</i>	90 <i>byte</i>	Branch
BCS <i>disp</i>	B0 <i>byte</i>	
BEQ <i>disp</i>	F0 <i>byte</i>	
BMI <i>disp</i>	30 <i>byte</i>	
BNE <i>disp</i>	D0 <i>byte</i>	
BPL <i>disp</i>	10 <i>byte</i>	
BVC <i>disp</i>	50 <i>byte</i>	
BVS <i>disp</i>	70 <i>byte</i>	
BIT <i>addr</i> <sub>0</sub>	24 <i>byte</i>	Bit test
BIT <i>addr</i>	2C <i>byte byte</i>	
BRK	00	Break
CLC	18	Clear Cs status
CLD	D8	Clear decimal status
CLI	58	Clear interrupt status
CLV	B8	Clear overflow status
CMP <i>#data</i>	C9 <i>byte</i>	Compare accumulator
CMP <i>addr</i> <sub>0</sub>	C5 <i>byte</i>	
CMP <i>addr</i>	CD <i>byte byte</i>	
CMP <i>addr</i> <sub>0</sub> ,X	D5 <i>byte</i>	
CMP <i>addr</i> ,X	DD <i>byte byte</i>	
CMP <i>addr</i> ,Y	D9 <i>byte byte</i>	
CMP ( <i>data</i> ,X)	C1 <i>byte</i>	
CMP ( <i>data</i> ),Y	D1 <i>byte</i>	
CPX <i>#data</i>	E0 <i>byte</i>	Compare register X
CPX <i>addr</i> <sub>0</sub>	E4 <i>byte</i>	
CPX <i>addr</i>	EC <i>byte byte</i>	
CPY <i>#data</i>	C0 <i>byte</i>	Compare register Y
CPY <i>addr</i> <sub>0</sub>	C4 <i>byte</i>	
CPY <i>addr</i>	CC <i>byte byte</i>	
DEC <i>addr</i> <sub>0</sub>	C6 <i>byte</i>	
DEC <i>addr</i>	CE <i>byte byte</i>	



**Table G-1—cont. 6502 Instruction Set**

<b>Mnemonic</b>	<b>Machine Language</b>	<b>Comments</b>
DEC <i>addr</i> <sub>0</sub> ,X DEC <i>addr</i> ,X DEX DEY	D6 <i>byte</i> DE <i>byte byte</i> CA 88	Decrement
EOR <i>#data</i> EOR <i>addr</i> <sub>0</sub> EOR <i>addr</i> EOR <i>addr</i> <sub>0</sub> ,X EOR <i>addr</i> ,X EOR <i>addr</i> ,Y EOR ( <i>data</i> ,X) EOR ( <i>data</i> ),Y	49 <i>byte</i> 45 <i>byte</i> 4D <i>byte byte</i> 55 <i>byte</i> 5D <i>byte byte</i> 59 <i>byte byte</i> 41 <i>byte</i> 51 <i>byte</i>	Logical EXCLUSIVE-OR
INC <i>addr</i> <sub>0</sub> INC <i>addr</i> INC <i>addr</i> <sub>0</sub> ,X INC <i>addr</i> ,X INX INY	E6 <i>byte</i> EE <i>byte byte</i> F6 <i>byte</i> FE <i>byte byte</i> E8 C8	Increment
JMP <i>addr</i> JMP ( <i>addr</i> ) JSR <i>addr</i>	4C <i>byte byte</i> 6C <i>byte byte</i> 20 <i>byte byte</i>	Jump
LDA <i>#data</i> LDA <i>addr</i> <sub>0</sub> LDA <i>addr</i> LDA <i>addr</i> <sub>0</sub> ,X LDA <i>addr</i> ,X LDA <i>addr</i> ,Y LDA ( <i>data</i> ,X) LDA ( <i>data</i> ),Y	A9 <i>byte</i> A5 <i>byte</i> AD <i>byte byte</i> B5 <i>byte</i> BD <i>byte byte</i> B9 <i>byte byte</i> A1 <i>byte</i> B1 <i>byte</i>	Load accumulator
LDX <i>#data</i> LDX <i>addr</i> <sub>0</sub> LDX <i>addr</i> LDX <i>addr</i> <sub>0</sub> ,Y LDX <i>addr</i> ,Y	A2 <i>byte</i> A6 <i>byte</i> AE <i>byte byte</i> B6 <i>byte</i> BE <i>byte byte</i>	Load X register

**Table G-1—cont. 6502 Instruction Set**

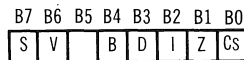
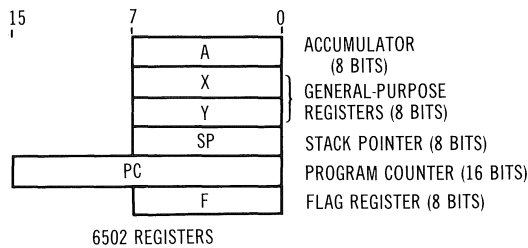
<b>Mnemonic</b>	<b>Machine Language</b>	<b>Comments</b>
LDY <i>#data</i>	A0 <i>byte</i>	Load Y register
LDY <i>addr</i> <sub>0</sub>	A4 <i>byte</i>	
LDY <i>addr</i>	AC <i>byte byte</i>	
LDY <i>addr</i> <sub>0</sub> ,X	B4 <i>byte</i>	
LDY <i>addr</i> ,X	BC <i>byte byte</i>	
LSR A	4A	Left Shift
LSR <i>addr</i> <sub>0</sub>	46 <i>byte</i>	
LSR <i>addr</i>	4E <i>byte byte</i>	
LSR <i>addr</i> <sub>0</sub> ,X	56 <i>byte</i>	
LSR <i>addr</i> ,X	5E <i>byte byte</i>	
NOP	EA	No operation
ORA <i>#data</i>	09 <i>byte</i>	Logical OR
ORA <i>addr</i> <sub>0</sub>	05 <i>byte</i>	
ORA <i>addr</i>	0D <i>byte byte</i>	
ORA <i>addr</i> <sub>0</sub> ,X	15 <i>byte</i>	
ORA <i>addr</i> ,X	1D <i>byte byte</i>	
ORA <i>addr</i> ,Y	19 <i>byte byte</i>	
ORA ( <i>data</i> ,X)	01 <i>byte</i>	
ORA ( <i>data</i> ),Y	11 <i>byte</i>	
PHA	48	Push accumulator to stack
PHP	08	Push flag register to stack
PLA	68	Load stack to accumulator
PLP	28	Load stack to flag register
ROL A	2A	Rotate left through carry
ROL <i>addr</i> <sub>0</sub>	26 <i>byte</i>	
ROL <i>addr</i>	2E <i>byte byte</i>	
ROL <i>addr</i> <sub>0</sub> ,X	36 <i>byte</i>	
ROL <i>addr</i> ,X	3E <i>byte byte</i>	
ROR A	6A	Rotate right through carry
ROR <i>addr</i> <sub>0</sub>	66 <i>byte</i>	
ROR <i>addr</i>	6E <i>byte byte</i>	
ROR <i>addr</i> <sub>0</sub> ,X	76 <i>byte</i>	
ROR <i>addr</i> ,X	7E <i>byte byte</i>	

**Table G-1—cont. 6502 Instruction Set**

<b>Mnemonic</b>	<b>Machine Language</b>	<b>Comments</b>
RTI	40	Return from interrupt
RTS	60	Return from subroutine
SBC <i>#data</i>	E9 <i>byte</i>	Subtract with carry (borrow)
SBC <i>addr</i> <sub>0</sub>	E5 <i>byte</i>	
SBC <i>addr</i>	ED <i>byte byte</i>	
SBC <i>addr</i> <sub>0</sub> ,X	F5 <i>byte</i>	
SBC <i>addr</i> ,X	FD <i>byte byte</i>	
SBC <i>addr</i> ,Y	F9 <i>byte byte</i>	
SBC ( <i>data</i> ,X)	E1 <i>byte</i>	
SBC ( <i>data</i> ),Y	F1 <i>byte</i>	
SEC	38	Set C's flag
SED	F8	Set decimal status
SEI	78	Set interrupt status
STA <i>addr</i> <sub>0</sub>	85 <i>byte</i>	Store accumulator
STA <i>addr</i>	8D <i>byte byte</i>	
STA <i>addr</i> <sub>0</sub> ,X	95 <i>byte</i>	
STA <i>addr</i> <sub>0</sub> ,X	9D <i>byte byte</i>	
STA <i>addr</i> ,Y	99 <i>byte byte</i>	
STA ( <i>data</i> ,X)	81 <i>byte</i>	
STA ( <i>data</i> ),Y	91 <i>byte</i>	
STX <i>addr</i> <sub>0</sub>	86 <i>byte</i>	Store X register
STX <i>addr</i>	8E <i>byte byte</i>	
STX <i>addr</i> <sub>0</sub> ,Y	96 <i>byte</i>	
STY <i>addr</i> <sub>0</sub>	84 <i>byte</i>	
STY <i>addr</i>	8C <i>byte byte</i>	
STY <i>addr</i> <sub>0</sub> ,X	94 <i>byte</i>	
TAX	AA	Transfer A to X
TAY	A8	Transfer A to Y
TSX	BA	Transfer SP to X

Table G-1—cont. 6502 Instruction Set

Mnemonic	Machine Language	Comments
TXA	8A	Transfer X to A
TXS	9A	Transfer X to SP
TYA	98	Transfer Y to A



6502 FLAG  
REGISTER DETAIL

- S. SIGN BIT
- V. OVERFLOW-STATUS BIT
- B. BREAK-STATUS BIT
- D. DECIMAL-STATUS BIT
- I. INTERRUPT-STATUS BIT
- Z. ZERO-STATUS BIT
- Cs. CARRY-STATUS BIT

Fig. G-1. Diagram of 6502 registers.

# Index

## A

- Absolute column-addressing, 40
- ADVANCE, 39, 73-74
- Advance cursor, 39-40
- Altering the character format, 77-80
- Alternative
  - character formats, 43-49
  - hi-res formats, 164-165
  - print windows, 50-62
  - screen formats, 134-140
- Analog
  - input clear, 250
  - inputs, 249
- APPEND, 252
- Assembler, 255
- Assembly language
  - and the miniassembler, 255-268
  - from machine language, 256-258
  - program preparation, 263-266
  - instruction set, 313-318

## B

- Backspace cursor, 40
- BASCALC, 73-74
- BASH, 72-73
- BASL, 72-73
- Binary, 269
  - to-decimal conversion, 274-275
  - to-hexadecimal conversion, 275-276
- BKGND routine, 153
- BRK instruction, 268
- BS routine, 206; *see also* monitor
- Building and using message blocks, 74-80
- Built-in memory mapped I/O, 243-250
- Byte, 269

## C

- CALL statement; *see also* monitor
  - 741, 206
  - 756, 214

## CALL statement—cont

- 868, 38
- 922, 40
- 926, 35
- 936, 37
- 998, 35
- 1008, 206
- 1036, 39
- 1994, 206
- 1998, 206
- 2008, 214
- 2023, 212
- 11465, 181
- 11471, 153
- 11500, 158
- 11506, 154
- 11527, 153
- 12274, 162
- 12288, 149

CALLing the INIT routine, 149-150; *see also* high-resolution graphics

Carriage return

- forced, 35
- suppressing, 21-23

Cassette

- IN jack, 248
- OUT, 245

CH register, 30, 73

Character codes; *see also* text codes

- flashing, 67
- inverse, 65-67

Clear keyboard strobe, 245

CLEAR routine, 162

Clear to end of; *see also* clearing

- line, 37-38
- page, 38-39

Clearing; *see also* clear to end of

- hi-res screen, 162
- screen, 27
- secondary page, 82-83

CLREOL routine, 38

CLREOP routine, 38-39

CLRSCR routine, 206  
 CLRTOP routine, 206  
 Color codes  
   full block, 125  
   hi-res, 156  
   low-resolution, 286; *see also* color codes  
     with COLOR statement  
   low-resolution upper/lower, 126-132, 287-300  
   with COLOR statement, 114; *see also* color codes, low-resolution  
 COLOR statement, 112-115  
 COLR variable; *see also* shape table  
   definition of, 147  
   values, 151  
 Column-addressing, absolute, 40  
 Column field, 23  
 Comma, 23  
 Controlling  
   cursor position with PRINT statements, 20-23  
   program flow with INPUT, 89-92  
 Conventional decimal to 2-byte decimal format, 272-273  
 Converting  
   large decimal values to smaller negative values, 273-274  
   negative decimal values to larger positive values, 274  
 COUT1 routine, 208-209  
 CR routine, 206; *see also* monitor  
 CTRL-B, 50  
 CTRL-C, 28  
 Cursor, 19  
   and CALL functions, 133  
   column address, 73  
   -positioning registers, 31-36  
     getting help from, 132-134  
 CV register, 30

## D

Debugging with BRK instruction, 268  
 Decimal-to-hexadecimal conversions, 271-272  
 Decoding single keystrokes for control purposes, 104-110  
 Defining the hi-res variables, 147-149  
 Disassembler, 256  
 Disassembly command, 257  
 Downward linefeed, 40  
 DRAW routine, 181  
 Drawing  
   horizontal lines with HLINE, 212-214  
   straight lines, 158-161  
   vertical lines with VLINE, 214

## E

ESC-A, 39  
 ESC-C, 40  
 ESC-D, 41  
 ESC-E, 38  
 ESC-, 27  
 Expansion ROM space \$C800-\$CFFF, 252

## F

Flashing  
   character codes, 67-68; *see also* flashing text codes  
   text codes, 68, 278; *see also* flashing character codes  
   text format, 43  
 Forced linefeed and carriage return, 35  
 48K systems, 241  
 Full-screen graphics, 135-138, 164-165

## G

Game controller  
   potentiometer, 249  
   socket, 246  
 Getting help from  
   cursor registers, 132-134  
   monitor, 70-74  
 GR statement, 111-112  
 Graphics, high-resolution  
   and DOS, 141  
   and HIMEM and LOMEM, 141-146  
   color codes, 156  
   full screen, 164-165  
   initializing, 147-150  
   programmer's aid routines, 141  
   secondary page, 165, 188-193  
   software switches, 164  
   variables, 147  
 Graphics, low-resolution  
   and cursor registers, 132-134  
   color codes, 120  
   full-screen, 135-138  
   primary page, 119  
   secondary page, 138-140  
   software switches, 134  
   techniques, 117-118

## H

Hand assembly, 255  
 Hexadecimal  
   numbers, 269  
   -to-binary conversion, 276  
   -to-decimal conversions, 270-271  
 Hi-res shape tables, 166-183; *see also* high resolution graphics

High-resolution graphics  
 and DOS, 141  
 and HIMEM and LOMEM, 141-146  
 background color, 151-153  
 BKGND routine, 153  
 CLEAR routine, 162  
 clearing the screen, 162  
 color codes, 156  
 colors and screen format, 150-151  
 full screen, 164-165  
 INIT routine, 149  
 initializing, 147-150  
 line drawing, 158-161  
 LINE routine, 158  
 plot coordinates, 153-158  
 PLOT routine, 154  
 POSN routine, 153  
 primary page, 183-188  
 programmer's aid routines, 141  
 secondary page, 165, 188-193  
 shape tables, 166-183  
 variables, 147  
 video addresses, 183-193  
 without shape tables, 151-163

## HIMEM

and high-resolution graphics, 141-146  
 and Integer BASIC, 237  
 programming of, 144-146  
 settings, 142-145

HIMEMH, 146

HIMEML, 146

HLIN statement, 115-117

HLINE routine, 212-214

Home, 36

HOME, 37

Home cursor and clear screen, 37

Homing cursor, 36-37

How this book is organized, 12-14

How to get most from this book, 14-15

## I

INIT routine, 147

Initializing the hi-res system, 147-150

INPUT statement

and menus, 90-92

and program flow, 89-92

and question marks, 86-89

and yes/no decisions, 89-90

syntax of, 86

Integer BASIC programming mode, 19

Inverse

character codes, 65-67; *see also* inverse  
 text codes

text codes, 67, 277

text format, 43

## I/O

addresses \$C000-\$CFFF, 242-252

port-0 slot, 85

port-1 slot, 85

## K

Key codes; *see also* keyboard character  
 codes

control keys, 285

ordinary keys, 282-284

Keyboard

character codes, 93-96; *see also* key codes

status input, 244-245

-to-video link, 85

KEYIN routine, 206

## L

LF routine, 40, 206; *see also* linefeed

LINE routine, 158

Linefeed; *see also* LF routine

downward, 40

forced, 35

suppressing, 21-23

upward, 35, 40-41

Loading through the

miniassembler, 265-266

monitor, 267

LOMEM, 81

and high-resolution graphics, 141-146

and Integer BASIC, 237

programming, 144-146

settings, 142-144

LOMEMH, 146

LOMEML, 146

Loudspeaker Toggle, 245-246

Low RAM addresses \$0000-\$0BFF, 225-235

## M

Machine language

calling a subroutine with, 199

disassembly of, 256-258

examples of, 200-202

instruction set, 313-318

loading data with, 196-198

loading of, 202-205

passing variables from, 223-224

passing variables to, 215-223

returning from a subroutine with, 199-200

running of, 205-206

storing data with, 198-199

Main ROM addresses \$D000-\$FFFF, 252-253

Memory map

of built-in I/O, 244

of I/O, 243

## Memory map—cont

- of primary page
  - hi-res graphics, 183-188
  - low resolution graphics, 199
  - text, 64
- of ROM, 253
- of secondary page
  - hi-res graphics, 188-193
  - low-resolution graphics, 120
  - text, 66
- of upper RAM
  - for 16K systems, 238
  - for 32K systems, 240
  - for 48K systems, 242
- Menus, 90-92
- Message blocks, 74-80
- Miniassembler, 255, 258-263
  - entering, 259
  - getting out of, 261
  - loading programs through, 259-261, 265-266
  - running a program from, 261
  - saving and loading tapes from, 262-263
- Mixing text formats, 47-49
- Monitor, 70-74
  - and low-resolution graphics, 118
  - BS routine, 206
  - CLRSCR routine, 206
  - CLRTOP routine, 206
  - COUT1 routine, 208-209
  - CR routine, 206
  - getting help from, 70-74
  - KEYIN routine, 206
  - LF routine, 206
  - loading programs through, 267
  - PRBL2 routine, 209
  - RDKEY routine, 214-215
  - routine CALLing, 206-215
  - routines available to BASIC, 206
  - routines not available to BASIC, 216-217
  - running a program from, 261-262
  - SETCOL routine, 210
  - STOADV routine, 206-208
  - UP routine, 206
- More cursor-related operations, 36-41
- MUSIC routine, 252

## N

- Nibble, 269
- Normal
  - low-resolution graphics, 111-112
  - text
    - codes, 67-70
    - format, 44
- NORMAL-1 text codes, 69, 278
- NORMAL-2 text codes, 69, 279

## O

- Organization of
  - low-resolution video memory, 119-122
  - text memories, 63-65

## P

- Passing variables
  - from machine-language routine, 223-224
  - to machine-language routine, 215-223
- PEEK statements
  - and menus, 106-108
  - and registers CH and CV, 31-33
  - and resuming stopped operations, 97-98
  - and stopping ongoing operations, 98-100
  - and strobing the keyboard, 92-97
  - and toggling operations, 100-102
  - and yes/no decisions, 105-106
- PEEKing into CH and CV, 31-33
- Peripheral card
  - I/O, 250-251
  - ROM, 251-252
- Peripheral slot scratchpad RAM, 235
- PLOT routine, 154
- PLOT statement
  - in BASIC, 112-115
  - in monitor, 210-212
- POKE statement
  - and CH register, 33-36
  - and CV register, 33-36
  - and low-resolution color graphics, 118-134
- POKEing
  - characters to secondary page, 83-84
  - into CH and CV, 33-36
- Port-0 I/O slot, 85
- Port-1 I/O slot, 85
- POSN routine, 153
- PRBL2 routine, 209
- Preparing assembly-language programs, 263-266
- Primary page
  - hi-res memory map, 183-188, 301-306
  - text and low-resolution graphics
    - displaying, 81
    - memory map, 64, 234, 280-281
    - uses, 233-234
- PRINT statement, 20, 29
- Program menus, 90-92
- Programmer's Aid routines, 141
  - APPEND, 252
  - BKGND (background color), 153
  - CLEAR (clearing the hi-res screen), 162
  - DRAW (shape-drawing), 181
  - HIGH-RESOLUTION GRAPHICS, 252
  - INIT (initialize primary page hi-res graphics), 149
  - LINE (line-drawing), 158



Programmer's Aid routines—cont

MUSIC, 252  
PLOT (point-plotting), 154  
POSN (point-positioning), 153  
RAM TEST, 252  
RENUMBER, 252  
TAPE VERIFY, 252

PR#1 command, 85

PR#2 command, 85

Pushbutton inputs PB1-PB3, 248

R

RAM, 225

keyboard input buffer, 231-233  
lower addresses \$0000-\$0BFF, 225-235  
peripheral slot scratchpad, 234  
primary page text/graphics, 233-234  
system stack, 231  
TEST, 252  
unused in secondary text page, 236  
upper addresses \$0C00-\$BFFF, 235-242  
variables, vectors, and user, 233  
zero-page, 226-231

RDKEY routine, 214-215

Registers

A, 195  
and low-resolution graphics, 132-134  
CH, 30  
cursor-positioning, 31-36  
CV, 30  
6502, 318  
X, 195  
Y, 195

RELOCATE, 252

RENUMBER, 252

Resuming stopped operations, 97-98

Role of

ADVANCE and BASCALC, 73-74  
cursor, 19-20  
BASL and BASH, 72-73  
CH, 73

ROM, 225

ROT

definition of, 147  
values, 180

SCALE variable, 147; *see also* shape table

Scratchpad addresses, 219

Screen mode switches, 134-135

SCRN statement, 117

Secondary page

hi-res memory map, 188-193, 306-312  
high-resolution graphics, 165  
text and low-resolution graphics  
clearing, 82-83  
definition of, 63

Secondary page—cont

*text and low-resolution graphics*  
displaying, 81  
memory map, 66, 236, 281  
POKEing characters to, 83-84  
unused RAM in, 236  
uses, 234-235  
working with, 80-84

Semicolon, 21

Set

full or mixed-screen text/graphics, 247  
graphics or text, 247  
or clear AN0-AN3 outputs, 247-248  
primary or secondary page, 247  
text/low-resolution or high-resolution, 247

SETCOL routine, 210

Setting

columns with commas, 23  
cursor position with TAB statements, 24-30  
number of characters per line, 51-53  
position of top line, 53-55  
position of the bottom line, 55-57  
starting column of text, 50-51  
text formats  
from keyboard, 44-45  
within a program, 46

Shape table

acronyms, 168  
codes, 168  
definition of, 166  
index of, 171-175  
loader, 175-177  
preparing data for, 168-171  
requirements for BASIC main program,  
177-183  
ROT values, 180  
setting up from BASIC, 167  
starting address, 178

SHAPE variable, 147; *see also* shape table

Simple PRINT statements, 20-21

Simplifying CALLs, 162-163

Simulating a PRINT @ statement, 29-30

Single-keystroke control of a program, 97-104

16K systems, 237-239

6502 instruction set, 313-318

6502 registers, 318

Split-screen, 59-62

Standard text format, 17-19

STEP, 268

STOADV, routine, 206-208

Stopping ongoing operations, 98-100

STRING\$ function, 22

Strobing keyboard with PEEK statements,  
92-97

Supplying information with INPUT, 85-89

Suppressing linefeed and carriage return,  
21-23  
with comma, 23

Suppressing linefeed and carriage return—cont Using  
 with semicolon, 21  
 Switching between primary and secondary  
 pages, 81-82

## T

TAB statement, 24-25  
 TAPE VERIFY, 252  
 Text  
   codes; *see also* character codes  
     inverse, 67  
     flashing, 68  
     normal, 67-70  
     NORMAL-1, 69, 278  
     NORMAL-2, 69, 279  
   editor  
     improved, 108-110  
     primitive, 96-97  
   format, 17  
     inverse, 43  
     flashing, 43  
     normal, 44  
     mixing, 47-49  
     setting, 44-46  
   software switches, 134  
   windows, 50-62  
 TEXT statement, 59, 111-112  
 32K systems, 239-241  
 Toggling operations, 100-101  
 TRACE, 268  
 Two-byte decimal to conventional decimal  
   format, 273

## U

Unused RAM in secondary text page, 236  
 UP routine, 40-41, 206  
 Upper RAM addresses \$0C00-\$BFFF, 235-  
   242  
 Upward linefeed, *see* UP routine

Utility Strobe output, 246

## V

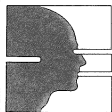
Video character codes, 65-70; *see also* char-  
   acter codes and text codes  
 VLIN statement, 115-117  
 VLINE routine, 214  
 VTAB statement, 25-27

## W

Which system do you need, 12  
 WNDBTM  
   definition of, 55, 57  
   values, 55, 57  
 WNDLFT  
   definition of, 50, 57  
   values, 52, 57  
 WNDRT, 53  
 WNDTOP  
   definition of, 53, 57  
   values, 55, 57  
 WNDWDTH  
   definition of, 51, 57  
   values, 52, 57  
 Working with  
   cursor-positioning registers, 31-36  
   secondary low-resolution graphics page,  
     138-140  
   secondary text page, 80-84

## X, Y

XX variable, 147; *see also* shape table  
 YY variable, 147; *see also* shape table



# SAMS APPLE® BOOKS

Many thanks for your interest in this Sams Book about Apple II® microcomputing. Here are a few more Apple-oriented Sams products we think you'll like:

## POLISHING YOUR APPLE®

Clearly written, highly practical, concise assembly of all procedures needed for writing, disk-filing, and printing programs with an Apple II. Positively ends your searches through endless manuals to find the routine you need! By Herbert M. Honig. 80 pages, 5½ x 8½, comb. ISBN 0-672-22026-1.

© 1982.

Ask for No. 22026. .... \$4.95

## THE APPLE II® CIRCUIT DESCRIPTION

Provides you with a detailed circuit description of the Revision 1 Apple II motherboard, including the keyboard and power supply. Compares Revision 1 with other revisions, and includes timing diagrams for major signals. By Winston D. Gayler. 176 pages plus foldouts, 8½ x 11, comb. ISBN 0-672-21959-X.

© 1983.

Ask for No. 21959. .... \$22.95

## INTERMEDIATE LEVEL APPLE II® HANDBOOK

Hands-on aid for exploring the entire internal firmware of your Apple II and finding out what you can accomplish with its 6502 microprocessor through machine- and assembly-language programming. By David L. Heiserman. 328 pages, 6 x 9, comb. ISBN 0-672-21889-5. © 1983.

Ask for No. 21889. .... \$16.95

## APPLE® FORTRAN

Only fully detailed Apple FORTRAN manual on the market! Ideal for Apple programmers of all skill levels who want to try FORTRAN in a business or scientific program. Many ready-to-run programs provided. By Brian D. Blackwood and George H. Blackwood. 240 pages, 6 x 9, comb. ISBN 0-672-21911-5. © 1982.

Ask for No. 21911. .... \$14.95

## APPLE II® ASSEMBLY LANGUAGE II

Shows you how to use the 3-character, 56-word vocabulary of Apple's 6502 to create powerful, fast-acting programs! For beginners or those with little or no assembly language programming experience. By Marvin L. De Jong. 336 pages, 5½ x 8½, soft. ISBN 0-672-21894-1. © 1982.

Ask for No. 21894. .... \$15.95

## ENHANCING YOUR APPLE II® — Vol. 1

Shows you how to mix text, LORES, and HIRES anywhere on the screen, how to open up whole new worlds of 3-D graphics and special effects with a one-wire modification, and more. Tested goodies from a trusted Sams author! By Don Lancaster. 232 pages, 8½ x 11, soft. ISBN 0-672-21846-1. © 1982.

Ask for No. 21846. .... \$15.95

## CIRCUIT DESIGN PROGRAMS FOR THE APPLE II® II

Programs quickly display "what happens if" and "what's needed when" as they apply to periodic waveform, rms and average values, design of matching pads, attenuators, and heat sinks, solution of simultaneous equations, and more. By Howard M. Berlin. 136 pages, 8½ x 11, comb. ISBN 0-672-21863-1. © 1982.

Ask for No. 21863. .... \$15.95

## APPLE® INTERFACING II

Brings you real, tested interfacing circuits that work, plus the necessary BASIC software to connect your Apple to the outside world. Lets you control other devices and communicate with other computers, modems, serial printers, and more! By Jonathan A. Titus, David G. Larsen, and Christopher A. Titus. 208 pages, 5½ x 8½, soft. ISBN 0-672-21862-3. © 1981.

Ask for No. 21862. .... \$10.95

**INTIMATE INSTRUCTIONS IN INTEGER BASIC**

Explains flowcharting, loops, functions, graphics, variables, and more as they relate to Integer BASIC. Used with *Applesoft Language* (No. 21811), it gives you everything you need to program BASIC with your Apple II or Apple II Plus. By Brian D. Blackwood and George H. Blackwood. 160 pages, 5½ x 8½, soft. ISBN 0-672-21812-7. © 1981.

**Ask for No. 21812.** .....\$8.95

**APPLESOFT® LANGUAGE**

Only complete text available on Applesoft BASIC! Self-teaching format simplifies learning and lets you use what you learn FAST. Ideal for businessmen, hobbyists, and professionals! Many programs included. By Brian D. Blackwood and George H. Blackwood. 256 pages, 5½ x 8½, soft. ISBN 0-672-21811-9. © 1981.

**Ask for No. 21811.** .....\$10.95

**MOSTLY BASIC: APPLICATIONS FOR YOUR APPLE II®, BOOK 1**

Twenty-eight debugged, fun-and-serious BASIC programs you can use immediately on your Apple II. Includes a telephone dialer, digital stopwatch, utilities, games, and more. By Howard Berenbon. 160 pages, 8½ x 11, comb. ISBN 0-672-21789-9. © 1980.

**Ask for No. 21789.** .....\$12.95

**MOSTLY BASIC: APPLICATIONS FOR YOUR APPLE II®, BOOK 2**

A second gold mine of fascinating BASIC programs for your Apple II, featuring 3 dungeons, 11 household programs, 6 on money or investment, 2 to test your ESP level, and more — 32 in all! By Howard Berenbon. 224 pages, 8½ x 11, comb. ISBN 0-672-21864-X. © 1981.

**Ask for No. 21864.** .....\$12.95

You can usually find these Sams products at better computer stores, bookstores, and electronic distributors nationwide.

If you can't find what you need, call Sams at 800-428-3696 toll-free or 317-298-5566, and charge it to your MasterCard or Visa account. Prices subject to change without notice.

For a free catalog of all Sams Books available, write P.O. Box 7092, Indianapolis IN 46206.

**SAMS BRINGS YOU MIND TOOLS™ FOR FINANCIAL PLANNING IN BUSINESS**

Special, ready-to-use software that temporarily interlocks with the spreadsheet in your regular version of Multiplan® or VisiCalc® so you can immediately perform 17 common financial planning calculations without wasting time manually setting up the sheet. All you do is enter the data — the proper formulas and column headings are there automatically!

Mind Tools allow you to instantly calculate present, net present, and future values, yields, internal and financial management rates of return, and basic statistics.

Also lets you do break-even analyses, depreciation schedules, and amortization tables, as well as compute variable- and graduated-rate mortgages, wraparound mortgages, and more!

Allows you to use your regular spreadsheet as you always have, at any time. Ideal for any businessman with financial planning responsibilities, as well as for business students and instructors.

Supplied with complete documentation, including 136-page text and 68-page quick-reference guide, all in a binder with the proper disk to match the brand of spreadsheet program you own.

Currently available for use with Multiplan or VisiCalc on the Apple II as follows:

**EXECUTIVE PLANNING WITH MULTIPLAN**

Apple II Version, ISBN 0-672-22058-X.

**Ask for No. 22058.** .....\$79.95

**EXECUTIVE PLANNING WITH VISICALC**

Apple II Version, ISBN 0-672-22059-8.

**Ask for No. 22059.** .....\$79.95



## TO THE READER

Sams Computer books cover Fundamentals — Programming — Interfacing — Technology written to meet the needs of computer engineers, professionals, scientists, technicians, students, educators, business owners, personal computerists and home hobbyists.

*Our Tradition is to meet your needs  
and in so doing we invite you to tell us what  
your needs and interests are by completing  
the following:*

1. I need books on the following topics:

---

---

---

---

---

---

2. I have the following Sams titles:

---

---

---

---

---

---

3. My occupation is:

<input type="checkbox"/> Scientist, Engineer	<input type="checkbox"/> D P Professional
<input type="checkbox"/> Personal computerist	<input type="checkbox"/> Business owner
<input type="checkbox"/> Technician, Serviceman	<input type="checkbox"/> Computer store owner
<input type="checkbox"/> Educator	<input type="checkbox"/> Home hobbyist
<input type="checkbox"/> Student	Other <input type="text"/>

Name (print)

Address

City  State  Zip

Mail to: **Howard W. Sams & Co., Inc.**

Marketing Dept. #CBS1/80  
4300 W. 62nd St., P.O. Box 7092  
Indianapolis, Indiana 46206





# **INTERMEDIATE-LEVEL APPLE II® HANDBOOK**

- Enjoy your Apple II® more by knowing how to use it better.
- Learn about nearly all configurations of your Apple II®.
- Combine machine-language and BASIC programs!
- Master useful programming techniques.
- Discover computer graphics techniques.
- Learn how to use the Mini-assembler.
- Do more with BASIC!
- Learn monitor routines.
- Debug programs!

**HOWARD W. SAMS & CO., INC.**

4300 West 62nd Street, Indianapolis, Indiana 46268 USA



# **INTERMEDIATE-LEVEL APPLE II® HANDBOOK**

**HEISERMAN •**

**21889**



**™**